USENIX

DISTRIBUTED AND MULTIPROCESSOR SYSTEMS WORKSHOP

AUTUMN 1989

USENIX

WORKSHOP PROCEEDINGS

DISTRIBUTED AND MUTIPROCESSOR SYSTEMS

October 5-6, 1989
Fort Lauderdale, FL

For additional copies of these proceedings write

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $30

Outside the U.S.A and Canada, please add
$20 per copy for postage (via air printed matter).

Copyright © 1989 USENIX Association
All Rights Reserved

This volume is published as a collective work.
Rights to individual papers remain
with the author or the author's employer.

UNIX is a registered trademark of AT&T.
Other trademarks are noted in the text.

Program and Table of Contents

Workshop on
Experiences with Distributed and Multiprocessor Systems

Fort Lauderdale, FL
October 5-6, 1989

Sponsored by the USENIX Association and the Software Engineering Research Center (SERC), in cooperation with ACM SIGOPS and ACM SIGSOFT, and with the IEEE-CS TC on OS and IEEE-CS TC on Distributed Systems.

Wednesday, October 4

Registration and Get-together 6:00 - 8:00

Thursday, October 5

Opening Remarks 8:30 - 8:45
George Leach, Paradyne Corp., Workshop Chair

Session I: Objects and Virtual Memory 8:45 - 10:00

A Distributed Implementation of the Shared Data-Object Model 1
Henri E. Bal, M. Frans Kaashoek and Andrew S. Tanenbaum
(Vrije Universiteit, Amsterdam)

An Implementation of Distributed Shared Memory 21
Umakishore Ramachandran and M. Yousef A. Khalidi
(Georgia Institute of Technology, Atlanta)

An Object-Oriented Implementation of Distributed Virtual Memory 39
Gary M. Johnston and R. H. Campbell
(University of Illinois at Urbana-Champaign)

Break 10:00 - 10:45

Session II: Process Control 10:45 - 12:00

Experience with Process Migration in Sprite 59
Fred Douglis (University of California, Berkeley)

Dynamic Server Squads in Yackos 73
Debra Hensgen and Raphael Finkel (University of Kentucky, Lexington)

Fine-Grain Scheduling 91
Henry Massalin and Calton Pu (Columbia University, New York)

Lunch 12:00 - 1:30

Session III: Performance Considerations 1:30 - 2:45

The Parallelization of Mach/4.3BSD:
Design Philosophy and Performance Analysis 105
Joseph Boykin and Alan Langerman
(Encore Computer Corp., Marlborough)

Efficient Implementation of Modularity in RAID 127
Charles Koelbel, Fady Lamaa, and Bharat Bhargava
(Purdue University, West Lafayette)

Making libc Suitable for use by Parallel Programs 145
Julie Kucera (Convex Computer Corp., Richardson)

Break 2:45 - 3:30

Session IV: Concepts 3:30 - 4:20

Revolution 89, or
Distributing UNIX Brings it Back to its Original Virtues 153
Francois Armand, Michel Gien, Frederic Herrmann, and Marc Rozier
(Chorus Systems, En Yvelines)

Experiences with the Distributed Data Structure Paradigm in Linda 175
M. Frans Kaashoek, Henri E. Bal, and Andrew S. Tanenbaum
(Vrije Universiteit)

Work-in-Progress presentations 4:20

Reception 6:00 - 8:00

Friday, Oct. 6

Session V: Multiprocessors 8:30 - 9:45

TUMULT-64: A Real-time Multi-processor System 193
Pierre G. Jansen and Gerard J. M. Smit
(University of Twente, Enschede)

Experiences with a Family of
Multiprocessor Real-Time Operating Systems 205
Prabha Gopinath and Thomas Bihari
(Philips Laboratories, Briarcliff Manor)

Implementation Issues for the
Psyche Multiprocessor Operating System 227
Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh
(University of Rochester)

Break 9:45 - 10:30

Session VI: Tools 10:30 - 12:15

Experience with P/Mothra:
A Tool for Mutation Based Testing on A Hypercube 237
ByoungJu Choi and Aditya P. Mathur
(Purdue University, West Lafayette)

Debugging and Performance Monitoring in HPC/VORX 255
Howard P. Katseff (AT&T Bell Laboratories, Holmdel)

CAPS — A Coding Aid used with the PASM Parallel Processing System . 269
James E. Lumpp, Jr., Samuel A. Fineberg, Wayne G. Nation,
Thomas L. Casavant, Edward C. Bronson, Howard J. Siegel,
Perre H. Pero, Thomas Schwederski, and Dan C. Marinescu
(Purdue University, West Lafayette)

The Implementation of Aide:
A Support Environment for Distributed Object-Oriented Systems 289
Rodger Lea and Johnathan Walpole (University of Lancaster, Bailrigg)

Lunch 12:15 - 1:30

Session VII: Object-oriented Construction 1:30 - 2:45

Experience With Implementing and Using
An Object-Oriented, Distributed System 301
D. Decouchant, M. Riveill, C. Horn, and E. Finn (Bull-IMAG, Gieres)

Prototyping a Distributed Object-oriented OS on UNIX 311
Marc Shapiro (INRIA, Le Chesnay)

The Clouds Experience:
Building an Object-Based Distributed Operating System 333
C. J. Wilkenloh, U. Ramachandran, S. Menon, R. J. LeBlanc, M. Y. A
Khaldi, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabeu,
W. F. Appelbe, and M. Ahamad
(Georgia Institute of Technology, Atlanta)

Break 2:45 - 3:30

Session VIII: Communications, Heterogeneous Systems, and the A-word 3:30 - 4:50

Experiences with Efficient Interprocess Communication in Dune 349
Marc F. Pucci and James Alberi
(Bell Communications Research, Morristown)

Using Transputer Networks to Accelerate Communication Protocols 361
Horst Schaaser (Hewlett-Packard Laboratories, Bristol)

ARCADE:
A Platform for Heterogeneous Distributed Operating Systems 373
David L. Cohn, William P. Delaney, and Karen M. Tracey
(University of Notre Dame)

A Decentralized Real-Time Operating System
Supporting Distributed Execution of Ada Tasks 391
Roger K. Shultz
(Rockwell International-Collins Divisions, Cedar Rapids)

Closing Remarks 4:50

First Workshop on Experiences with Building Distributed and Multiprocessor Systems (WEBDMS)

5-6 October 1989
Ft. Lauderdale, FL

Sponsored by

The USENIX Association
The Purdue/University of Florida Software Engineering Research Center (SERC)

In cooperation with

ACM Special Interest Group on Operating Systems (SIGOPS)
ACM Special Interest Group on Software Engineering (SIGSOFT)
Computer Society of the IEEE, Technical Committee on Operating Systems (TCOS)
Computer Society of the IEEE, Technical Committee on Distributed Systems (TCDS)

Program Committee

George W. Leach (general chair)
AT&T Paradyne

Eugene H. Spafford (program chair)
Department of Computer Sciences
Purdue University

Bharat Bhargava
Department of Computer Sciences
Purdue University

Joseph Boykin
Encore Computer Corporation

Darrell Long
Computer Engineering & Information Sciences
University of CA, Santa Cruz

James Mankovich
Convex Computer Corporation

Eugene Miya
NASA Ames Research Center

David V. Pitts
Department of Computer Science
University of Lowell

Rob Kolstad
Prisma

Reviewers

George Adams
Bob Brown
Panayiotis Chrysanthis
Fred Douglis
James Griffioen
Kurt Horton
John Lekashman
Dean Long
Charlie McDowell
Thomas Narten
Gerry Pocock
Kenneth Rodemann
Keith Thompson
C. Thomas Wilkes

Arun Arya
Shirley Browne
Bob Ciotti
Ray Essick
P. Michael Haffley
Byung G. Kim
Guoben Li
Stephen Lundstrom
Bill Moloney
Mike O'Dell
Krithi Ramamritham
Tony Schene
Brent Welch
Pete Wilson

Keith Bierman
Steve Chapin
Al Conrad
Lilly J. Fuller
Abdesalam Helal
Charles Koelbel
Shy-Renn Lian
Enrique Mafla
Bruce Montague
A. Jeff Offutt
John Riedl
Stephen A. Shumway
Brent Welch
Michael J. Yamasaki

A DISTRIBUTED IMPLEMENTATION OF THE SHARED DATA-OBJECT MODEL

Henri E. Bal *
M. Frans Kaashoek
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Email: bal@cs.vu.nl

ABSTRACT

The shared data-object model is designed to ease the implementation of parallel applications on loosely coupled distributed systems. Unlike most other models for distributed programming (e.g., RPC), the shared data-object model allows processes on different machines to share data. Such data are encapsulated in data-objects, which are instances of user-defined abstract data types. The shared data-object model forms the basis of a new language for distributed programming, *Orca*, which gives linguistic support for parallelism and data-objects. A distributed implementation of the shared data-object model should take care of the physical distribution of objects among the local memories of the processors. In particular, an implementation may replicate objects in order to decrease access times to objects and increase parallelism.

The intent of this paper is to show that, for several applications, the proposed model is both easy to use and efficient. We first give a brief description of the shared data-object model and Orca. Next, we describe one of several existing implementations of Orca. This implementation replicates all objects on all processors and updates replicas through a reliable broadcast protocol. We describe all three layers of this implementation: the Orca compiler, the Orca run time system, and the reliable broadcast protocol. Finally, we report on our experiences in using this implementation. We describe three parallel applications written in Orca and give performance measurements for them. We also compare these figures with those of a nondistributed (shared-memory) implementation of Orca. The measurements show that significant speedups can be obtained for all three applications.

* This research was supported in part by the Netherlands organization for scientific research (N.W.O.) under grant 125-30-10.

1. INTRODUCTION

As communication in loosely coupled distributed computing systems is getting faster, such systems become more and more attractive for running parallel applications. In the Amoeba system [Mullender and Tanenbaum 1986], for example, the cost of sending a short message between Sun workstations over an Ethernet is 1.4 milliseconds [Van Renesse et al. 1989]. Although this is still slower than communication in most multicomputers (e.g., Hypercubes and transputer grids), it is fast enough for many coarse-grained parallel applications. In return, distributed systems are easy to build from off-the-shelf components, by interconnecting multiple workstations or microprocessors through a local area network (LAN). In addition, such systems can easily be expanded to far larger numbers of processors than shared-memory multiprocessors.

In our research, we are studying the implementation of parallel applications on distributed systems. We started out by implementing several coarse-grained parallel applications on top of the Amoeba system, using Remote Procedure Calls (RPC) [Birrell and Nelson 1984] for interprocess communication [Bal et al. 1987]. RPC is widely used in the distributed systems community for implementing distributed servers (e.g., file servers) [Tanenbaum and Van Renesse 1985]. For parallel programming, however, RPC has several disadvantages [Tanenbaum and Van Renesse 1988]. RPC is a synchronous (blocking) communication primitive, so a separate mechanism is needed for obtaining parallelism. Of more significance, the programming model of RPC is based on message passing, which is conceptually input/output. This makes efficient sharing of data among processes very hard.

The RPC model does not provide (logically) shared data, since processes on different machines run in separate address spaces. Data that are shared among multiple processes have to be encapsulated by a server process and can only be accessed indirectly through a remote call to this server. Parallel applications, however, often need a finer level of sharing, with a much lower overhead.

As an example of such a parallel application, consider parallel branch-and-bound algorithms. Such algorithms store the current best solution (the bound) in a global variable accessed by all processors. This is not to say the algorithms actually need physical shared memory; as the bound is updated only once in a while, parallel branch-and-bound algorithms can be implemented efficiently on distributed systems. In our experience, however, implementing the algorithms efficiently using RPC is complicated.

In this paper, we will look at an alternative model for distributed programming that supports logically shared data. This model, the *shared data-object model* [Bal and Tanenbaum 1988], allows processes to share data without requiring physical shared memory. Also, we have designed a new programming language, *Orca* [Bal and Tanenbaum 1988; Bal et al. 1989], based on this model. The intent of this paper is to show that, for several applications, the model is both easy to use and efficient. We do so by describing an implementation of Orca on a loosely coupled system and reporting on our experiences in using this implementation for several small-scale but realistic applications.

The issue of providing logically shared data in an environment without shared memory has been addressed by several other languages and operating systems. Linda's Tuple Space [Ahuja et al. 1986], for example, is a global, content-addressable shared memory, which has been implemented on various types of parallel systems. For many applications this model is much easier to use than RPC. The operations defined on Tuple Space provide a low level of abstraction, however, which we feel is a disadvantage for distributed programming [Kaashoek et al. 1989a].

Other interesting proposals include parallel object-oriented languages (e.g., Emerald [Jul et al. 1988]), which provide a uniform address space for objects, and Kai Li's shared virtual memory [Li 1988], which simulates physical shared memory. (These and other systems are surveyed in [Bal and Tanenbaum 1988].) Also, several researchers have looked at distributed applications that can be implemented with logically shared data. Example applications are: speech recognition [Bisiani and Forin 1987], linear-equation solving, three-dimensional partial differential equations [Li 1988], and global scheduling and replicated files [Cheriton 1985].

The rest of the paper is structured as follows. In Section 2, we will give a brief description of the shared data-object model and Orca. In Section 3, we will discuss one implementation of the model, based on reliable broadcast. We will also describe how to implement this broadcast primitive on top of LANs that only support unreliable broadcast. In Section 4, we will report on our experiences in using this implementation of Orca. We will give performance measurements for several applications. Also, we will compare these performance figures with those of a non-distributed (shared-memory) implementation of Orca. Finally, in Section 5 we present our conclusions.

2. THE SHARED DATA-OBJECT MODEL

The most important issue addressed by our model is how data structures can be shared among distributed processes in an efficient way. In languages for multiprocessors, shared data structures are stored in the shared memory and accessed in basically the same way as local variables, namely through simple load and store instructions. If a process is going to change part of a shared data structure and it does not want other processes to interfere, it locks that part. All these operations (loads, stores, locks) on shared data structures involve very little overhead, because access to shared memory is hardly more expensive than access to local memory.

In a distributed system, on the other hand, the time needed to access data very much depends on the location of the data. Accessing data on remote processors may be orders of magnitude more expensive than accessing local data. It is therefore infeasible to apply the multiprocessor model of programming to distributed systems. The operations used in this model are far too low-level and will have tremendous overhead on distributed systems.

The starting-point in our model is to access shared data structures through higher level operations. Instead of using low-level instructions for reading, writing, and locking shared data, we propose to let programmers define composite operations for manipulating shared data structures. Shared data structures in our model are encapsulated in so-called *data-objects*[1] that are manipulated through a set of user-defined operations. Data-objects are best thought of as instances (variables) of *abstract data types*. The programmer specifies an abstract data type by defining operations that can be applied to instances (data-objects) of that type. The actual data contained in the object and the executable code for the operations are hidden in the implementation of the abstract data type.

Although data-objects logically are shared among processes, their implementation does not need physical shared memory. In worst case, an operation on a remote object can be implemented with a remote procedure call. The general idea, however, is for the implementation to take care of the physical distribution of data-objects among processors. As we will see in

[1] We will sometimes use the term "object" as a shorthand notation. Note, however, that this term is used in many other languages and systems, with various different meanings.

Section 3, one way to achieve this goal is to replicate shared data-objects. By replicating objects, access control to shared objects is decentralized, which decreases access costs and increases parallelism. This is a major difference with, say, monitors [Hoare 1974], which centralize control to shared data.

In the following sections, we will elaborate the basic idea by looking at the issue of synchronization. Two types of synchronization can be distinguished [Andrews and Schneider 1983]: mutual exclusion synchronization prevents multiple simultaneous writes (or reads and writes) to the same data from interfering with each other; condition synchronization allows processes to wait for a certain condition to become true. We discuss both types of synchronization in turn, in Sections 2.1 and 2.2. Finally, in Section 2.3 we describe a language based on this model.

2.1. Mutual exclusion synchronization

Shared-variable languages usually provide some kind of *locking* construct for mutual exclusion synchronization. In a distributed environment, however, such locking primitives are too low-level and have a high overhead. In our model, mutual exclusion is done implicitly, by executing all operations on objects *indivisibly*. Conceptually, each operation locks the entire object it is applied to and releases the lock only when it is finished. To be more precise, the model guarantees *serializability* [Eswaran et al. 1976] of operation invocations: if two operations are applied simultaneously to the same data-object, then the result is as if one of them is executed before the other; the order of invocation, however, is nondeterministic.

An implementation of the model need not actually execute all operations one by one. To increase the degree of parallelism, it may execute multiple operations on the same object simultaneously, as long as the effect is the same as for serialized execution. For example, operations that only read (but do not change) the data stored in an object can easily be executed in parallel.

As operations are indivisible, mutual exclusion synchronization to shared data-objects is taken care of automatically. As a simple example, consider an object encapsulating an integer variable, as specified in Figure 1.

```
object specification IntObject;
  operation Value(): integer;          # return current value
  operation Assign(val: integer);      # assign new value
  operation Add(val: integer);         # add val to current value
  operation Min(val: integer);         # set value to minimum of current value and val
end;
```

Fig. 1. Specification part of an object type IntObject.

Suppose two processes P_1 and P_2 share an object X of this type. If they simultaneously try to apply the *Assign* operation to X, the resulting value will either be that of P_1's or P_2's invocation, but the value will never be some strange mixture of the bits. Similarly, if P_1 and P_2 simultaneously increment the value of X by invoking the operation

```
X$Add(1);
```

the value will always be incremented twice, because the operations are serialized.

On the other hand, *sequences* of operations are not executed indivisibly. For example, the sequence

```
tmp := X$Value();       # get value of object X
X$Assign(tmp+1);        # increment value and store result back in X
```

is not an indivisible action. If two processes execute this sequence simultaneously, the value of *X* may be incremented once or twice. This rule for defining which actions are indivisible and which are not is both easy to understand and flexible: single operations are indivisible; sequences of operations are not. Orca does not provide mutual exclusion at a granularity lower than the object level.

Our model does not support indivisible operations on multiple objects. Operations on multiple objects would require a distributed locking protocol, which is complicated to implement efficiently. Instead, we prefer to keep our basic model as simple as possible and implement more complicated actions on top of it. Operations in our model therefore apply to single objects and are always executed indivisibly. However, the model is sufficiently powerful to allow users to construct locks for multi-operation sequences on different objects, so arbitrary actions can be performed indivisibly.

2.2. Condition synchronization

Condition synchronization allows processes to wait (*block*) until a certain condition becomes true. The simplest form of condition synchronization is repeated testing (*busy waiting*) of a shared variable, until it has a certain value. Since busy waiting wastes computing cycles, most parallel languages use a separate condition synchronization mechanism, such as a semaphore, eventcount, or condition variable [Andrews and Schneider 1983].

In the shared data-object model, condition synchronization is integrated with operation invocations by allowing operations to block. Processes synchronize implicitly through operations on shared objects. A blocking operation consists of one or more guarded commands:

```
operation op(formal-parameters): ResultType;
  local declarations
begin
   guard condition_1 do statements_1 od;
   guard condition_2 do statements_2 od;
   ...
   guard condition_n do statements_n od;
end;
```

The conditions must be side-effect free boolean expressions. The operation initially blocks (suspends) until at least one of the conditions (guards) evaluates to "true." Next, one true guard is selected nondeterministically, and its sequence of statements is executed.

2.3. Orca

We have used the shared data-object model for designing a new language called *Orca* for distributed application programming. Unlike the majority of other languages for parallel or distributed programming, Orca is not an extension to an existing sequential language. Instead, its sequential and distributed constructs have been designed together, in such a way that they integrate well.

Orca is a procedural, strongly typed language. Its statements and expressions are fairly conventional and comparable to those of Modula-2. The data structuring facilities of Orca, however, are substantially different from those used in Modula-2. Orca supports records, unions, dynamic arrays, sets, bags, general graphs, and generic types. Pointers have intentionally been omitted to provide type-security.

Parallelism in Orca is based on explicit creation of sequential processes. Processes are conceptually similar to procedures, except that procedure invocations are serial and process invocations are parallel.

Processes communicate through shared data-objects, which are instances of *abstract data types*. An abstract data type definition consists of two parts: a *specification* part and an *implementation* part. The specification part defines the operations applicable to objects of the given type. (An example of a specification part was given in Figure 1.) The implementation part contains the data of objects of this type, the code to initialize the data of new instances of the type, and the code implementing the operations.

Objects are created by declaring variables of an abstract data type. The declaration does not specify whether the object will be shared. When an object is created, the run time system allocates memory for the local variables of the object and executes the initialization code.

Objects declared local to a process may be shared with other (child) processes by passing them as shared parameters when the children are created. For example, if a process *child* is declared as

process *child(Id: integer; X:* **shared** *IntObject);*

a new child process can be created as follows

```
fork child(12, X);
  # create a new child process, passing the constant 12 as
  # value parameter and the object X as shared parameter.
```

The children can pass shared objects to *their* children, and so on. In this way, the objects get distributed among some of the descendants of the process that created them. If any of these processes performs an operation on the object, they all observe the same effect, as if the object were in shared memory, protected by a lock variable.

In summary, Orca allows processes to share data encapsulated in objects, which are instances of abstract data types. Sharing of objects is only possible between a parent and its descendants, which is sufficient for the applications Orca intends to support. Each process sharing an object may apply operations to the object, as defined by the object's abstract data type. The effects of operation invocations are observed by all processes sharing the object. Simultaneous invocations of operations on the same object are conceptually serialized. Condition synchronization is expressed through operations that block.

3. A DISTRIBUTED IMPLEMENTATION OF ORCA

Although Orca is a language for programming distributed systems, its communication model is based on shared data. The implementation of the language therefore should hide the physical distribution of the hardware and simulate shared data in an efficient way. We have designed several different models for implementing the language [Bal and Tanenbaum 1988]. The implementation described in this paper is based on *replication* and *reliable broadcasting*.

Replication of data is used in several fault-tolerant systems (e.g., ISIS [Joseph and Birman 1987]) to increase the availability of data in the presence of processor failures. Orca, in contrast, is not intended for fault-tolerant applications. In our implementation, replication is used to decrease the access costs to shared data.

Very briefly stated, each processor keeps a local copy of each shared data-object. This copy can be accessed by all processes running on that processor (see Figure 2). Operations that do not change the object (called *read* operations) use this copy directly, without any messages being sent. Operations that do change the object (called *write* operations) broadcast the new values (or the operations) to all the other processors, so they are updated simultaneously.

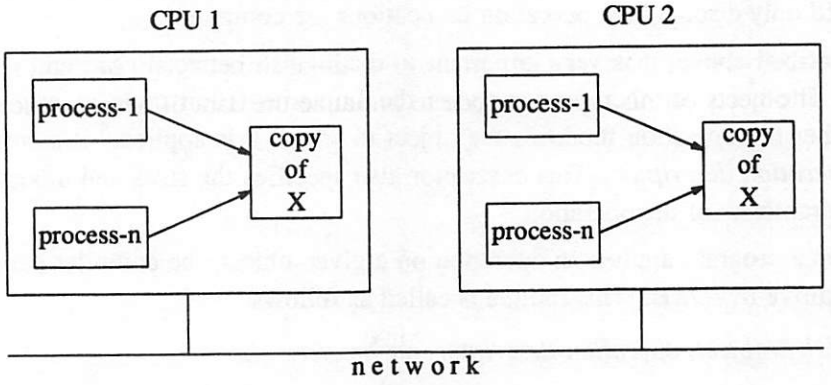


Fig. 2. Replication of data-objects in a distributed system

The implementation is best thought of as a three layer software system, as shown below:

compiled application programs
run time system
reliable broadcasting

The top layer is concerned with applications, which are written in Orca and compiled to machine code by the Orca compiler. The executable code contains calls to the Orca run time system, for example for creating and manipulating processes and objects.

The middle layer is the run time system (RTS). It implements the primitives called by the upper layer. For example, if an application performs an operation on a shared data-object, it is up to the RTS to ensure that the system behaves as if the object was placed in shared memory. To achieve this, the RTS of each processor maintains copies of shared objects, which are updated using reliable broadcasting.

The bottom layer is concerned with implementing the reliable broadcast, so that the RTS does not have to worry about what happens if a broadcast message is lost. As far as the RTS is concerned, broadcast is error free. It is the job of the bottom layer to make it work.

Below, we will describe the protocols and algorithms in each layer. This section is structured top down: we first discuss the applications layer, then the RTS layer, and finally the reliable broadcast layer.

3.1. Top layer: Orca application programs

Application programs are translated by the Orca compiler into executable code for the target system.[2] Most of the compiler is based on conventional compiler technology. In fact, our compiler has been built using the *Amsterdam Compiler Kit*, which is a toolkit for implementing portable compilers [Tanenbaum et al. 1983]. Up until now ACK has mainly been used for sequential languages like C and Pascal and for uniprocessor implementations of parallel (or pseudo-parallel) languages like Modula-2, occam, and Ada®. As it turns out, ACK is useful for distributed languages like Orca as well.

The code produced by the compiler contains calls to RTS routines that manage processes, shared data-objects, and complex data structures (e.g., dynamic arrays, sets, and graphs). In this paper, we will only discuss how operation invocations are compiled.

As described above, it is very important to distinguish between *read* and *write* operations on objects. The compiler therefore analyses the implementation code of each operation and checks whether the operation modifies the object to which it is applied.[3] It stores this information in an *operation descriptor*. This descriptor also specifies the sizes and modes (input or output) of the parameters of the operation.

If an Orca program applies an operation on a given object, the compiler generates a call to the RTS primitive *INVOKE*. This routine is called as follows:

INVOKE(object, operation-descriptor, parameters ...);

The first argument identifies the object to which the operation is applied. The second argument is the operation descriptor. The remaining arguments of *INVOKE* are the parameters of the operation. The implementation of this primitive is discussed below.

3.2. Middle layer: The Orca run time system

The middle layer implements the Orca run time system. As mentioned above, its primary job is to manage shared data-objects. In particular, it implements the *INVOKE* primitive described above. For efficiency, the RTS replicates objects so it can apply operations to local copies of objects whenever possible.

There are many different design choices to be made related to replication. The most important ones are:

Replication strategy:

The RTS may either replicate all objects on all processors (*full replication*) or it may try to replicate objects only on those processors that frequently read the object (*partial replication*). In the latter case, the RTS may use compile-time information as well as run-time statistics for deciding where to store replicas of objects.

Updating of replicas:

After a write operation, the replicas of an object should either be invalidated or updated. Updating can either be implemented by sending the new value of the object to the other processors or by applying the operation itself to each copy.

---

[2] We assume the target system does not contain multiple types of CPUs. Although a heterogeneous implementation of Orca is conceivable, we do not address this issue here.

[3] The actual implementation is somewhat more complicated, since an operation may have multiple guards (alternatives), some of which may be read-only.

Mutual exclusion synchronization:

Write operations on a given object can be synchronized in at least two different ways. One way is to appoint some copy of the object as *primary copy* and direct all write operations to this primary copy. An alternative way is to treat all copies as equals and use a *distributed update protocol* that takes care of mutual exclusion.

Each of these alternatives has its own advantages and disadvantages, as discussed in [Bal and Tanenbaum 1988]. The RTS described in this paper uses full replication of objects, updates replicas by applying write operations to all replicas, and implements mutual exclusion through a distributed update protocol. (We have also implemented a second RTS, which uses partial replication based on run-time statistics and which updates copies through a primary-copy update protocol. In addition, we have implemented a third RTS on a true shared-memory multiprocessor, for comparison purposes.)

We have chosen to use an update scheme rather than an invalidation scheme for two reasons. First, in many applications objects contain large amounts of data (e.g., a 100K bitvector). Invalidating a copy of such an object is wasteful, since the next time the object is replicated its entire value must be transmitted. Second, in many cases updating a copy will take just as much CPU time and network bandwidth as sending invalidation messages.

The presence of multiple copies of the same logical data introduces the so-called *inconsistency problem*. If the data are modified, all copies are modified too. If this updating is not done as one indivisible action, different processors temporarily have different values for the same logical data. (The inconsistency problem appears in many other areas where data are replicated, for example replicated file servers and CPU caches.)

The semantics of the shared data-object model define that simultaneous operations on the same object must conceptually be serialized. The exact order in which they are to be executed is not defined, however. If, for example, a read operation and a write operation are applied to the same object simultaneously, the read operation may either observe the value before or after the write, but not an intermediate value. However, all processes having access to the object must see the events happen in the same order.

The RTS described here solves the inconsistency problem by using a distributed update protocol that guarantees that all processes observe changes to shared objects *in the same order*. One way to achieve this would be to lock all copies of an object prior to changing the object. Unfortunately, distributed locking is quite expensive and complicated.

Our update protocol does not use locking. The key to avoid locking is the use of an *indivisible, reliable broadcast* primitive, which has the following properties:

- Each message is sent reliably from one source to all destinations.
- If two processors simultaneously broadcast two messages (say m_1 and m_2), then either all destinations first receive m_1, or they all receive m_2 first. Mixed forms (some get m_1 first, some get m_2 first) are excluded by the software protocols.

This primitive is implemented by the bottom layer of our system, as will be described in Section 3.3, Here, we simply assume the indivisible, reliable broadcast exists.

The RTS uses an *object-manager* for each processor. The object-manager is a light-weight process (thread) that takes care of updating the local copies of all objects stored on its processor. We assume the object-manager and user processes on the same processor can share part of their address space. Objects (and replicas) are stored in this shared address space. User processes

can *read* local copies directly, without intervention by object-managers. Write operations on shared objects, on the other hand, are marshalled and then broadcast to all object-managers in the system. A user process that broadcasts a write operation suspends until the message has been handled by its local object-manager. This is illustrated in Figure 3.

```
INVOKE(obj, op, parameters)
    if op.ReadOnly then                    # check if it's a read operation
        set read-lock on local copy of obj;
        call op.code(obj, parameters);     # do operation locally
        unlock local copy of obj
    else
        broadcast GlobalOperation(obj, op, parameters) to all managers;
        block current process;
    fi;
```

Fig. 3. Implementation of the *INVOKE* run time system primitive. This routine is called by user processes.

Each object-manager maintains a queue of messages that have arrived but that have not yet been handled. As all processors receive all messages in the same order, the queues of all managers are basically the same, except that some managers may be ahead of others in handling the messages at the head of the queue.

The object-manager of each processor handles the messages of its queue in strict FIFO order. A message may be handled as soon as it appears at the head of the queue. To handle a message *GlobalOperation(obj, op, parameters)* the message is removed from the queue, unmarshalled, the local copy of the object is locked, the operation is applied to the local copy, and finally the copy is unlocked. If the message was sent by a process on the same processor, the manager unblocks that process (see Figure 4).

```
receive GlobalOperation(obj, op, parameters) from W →
    set write-lock on local copy of obj;
    call op.code(obj, parameters);  # apply operation to local copy
    unlock local copy of obj
    if W is a local process then
        unblock(W);
    fi;
```

Fig. 4. The code to be executed by the object-managers for handling *GlobalOperation* messages.

Write operations are executed by all object-managers in the same order. If a read operation is executed concurrently with a write operation, the read may either be executed before or after the write, but not during it. Note that this is in agreement with the serialization principle described above.

3.3. Bottom layer: Reliable broadcast

In this section we describe the protocol that allows a group of nodes on an unreliable broadcast network to broadcast messages reliably. The protocol guarantees that all of the receivers in the group receive all broadcast messages and that all receivers accept the messages in the same order. The main purpose of this section is to show that a protocol with the required semantics is feasible; for a detailed description we refer the reader to [Kaashoek et al. 1989b].

With current microprocessors and LANs, lost or damaged packets and processor crashes occur very infrequently. Nevertheless, the probability of an error is not zero, so they must be dealt with. For this reason our approach to achieving reliable broadcast is to make the normal case highly efficient, even at the expense of making error-recovery more complex, since error recovery will not be done very often.

The basic reliable broadcast protocol works as follows. When the RTS wants to broadcast a message, M, it hands the message to its kernel. The kernel then encapsulates M in an ordinary point-to-point message and sends it to a special kernel called the *sequencer*. The sequencer's node contains the same hardware and kernel as all the others. The only difference is that a flag in the kernel tells it to process messages differently. If the sequencer should crash, the protocol provides for the election of a new sequencer on a different node.

The sequencer determines the ordering of all broadcast messages by assigning a *sequence number* to each message. When the sequencer receives the point-to-point message containing M, it allocates the next sequence number, s and broadcasts a packet containing M and s. Thus all broadcasts are issued from the same node, by the sequencer. Assuming that no packets are lost, it is easy to see that if two RTSs simultaneously want to broadcast, one of them will reach the sequencer first and its message will be broadcast to all the other nodes first. Only when that broadcast has been completed will the other broadcast be started. The sequencer provides a global ordering in time. In this way, we can easily guarantee the atomicity of broadcasting.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast packet eventually arrives, the kernel will immediately notice a gap in the sequence numbers. It was expecting s next, and it got $s + 1$, so it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for copies of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores old broadcast messages in its *history buffer*. The missing messages are sent point-to-point to the process requesting them.

As a practical matter, the sequencer has a finite amount of space in its history buffer, so it cannot store broadcast messages forever. However, if it could somehow discover that all machines have received broadcasts up to and including k, it could then purge the first k broadcast messages from the history buffer.

The protocol has several ways of letting the sequencer discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message. In this way, the sequencer can maintain a table, indexed by node number, showing that node i has received all broadcast messages 0 up to T_i, and perhaps more. At any moment, the sequencer can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, the

sequencer knows that everyone has received broadcasts 0 through 6, so they can be deleted from the history buffer.

If a node does not need to do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval, Δt, can just send the sequencer a special packet acknowledging all received broadcasts.

If, despite all precautions, the sequencer gets out of history space, it enters a synchronization phase to empty its history buffer. The synchronization phase consists of a two-phase commit protocol, during which all nodes are brought up-to-date. In practice, the synchronization phase is hardly ever entered.

In short, to do a broadcast, an application process sends the data to the sequencer, which gives it a sequence number and broadcasts it. There are no separate acknowledgement packets, but all messages to the sequencer carry piggybacked acknowledgements. When a node receives an out of sequence broadcast, it buffers the broadcast temporarily, and asks the sequencer for the missing broadcasts. Since broadcasts are expected to be common—many per second—the only effect that a missed broadcast has is causing some application process to get behind by a few tens of milliseconds once in a while, hardly a serious problem.

In philosophy, the protocol resembles the one described by Chang and Maxemchuk [Chang and Maxemchuk 1984], but differs in some major aspects. Messages can be delivered to the user as soon as one (special) node has acknowledged the message. In addition, fewer control messages are needed in the normal case (no lost messages). Our protocol therefore is highly efficient, since, during normal operation, only two packets are needed (assuming that a message fits in a single packet), one point-to-point packet from the sender to the sequencer and one broadcast packet from the sequencer to everyone. A comparison between our protocol and other well known protocols (e.g., those of Birman and Joseph [Birman and Joseph 1987], Garcia-Molina and Spauster [Garcia-Molina and Spauster 1989], and several others) is given in [Kaashoek et al. 1989b].

4. EXPERIENCE WITH THE ORCA IMPLEMENTATION

We have built a prototype implementation of the shared data-object model, using the layered approach described in the previous section. The prototype runs on the bare hardware, rather than on top of an operating system. In effect, it is a new kind of operating system designed specifically for parallel applications. It uses the Amoeba protocols [Mullender and Tanenbaum 1986] to communicate with our local UNIX® and Amoeba systems.

The prototype runs on two different systems. One implementation runs on a multiprocessor with 10 16 Mhz MC68020 CPUs. The system contains 8Mb of shared memory, which is accessible through a VME bus. This implementation uses the shared memory to simulate unreliable broadcast messages. The reliability of the network (i.e., the percentage of broadcast messages delivered at a destination) is an adjustable parameter of the system. In this way, we are able to test our protocol with different degrees of reliability. The second implementation runs on a distributed system, containing 10 16 Mhz MC68020 CPUs connected to each other through an 10 Mbit/s Ethernet [Metcalfe and Boggs 1976]. This implementation uses Ethernet multicast communication to broadcast a message to a group of processors. All processors are on one Ethernet and are connected to the network by Lance chip interfaces.

The performance of the broadcast protocol on the Ethernet system is described in [Kaashoek et al. 1989b]. The time needed for multicasting a short message reliably to two processors is 1.3 msec. With 10 receivers, a multicast takes 1.5 msec. The time also depends on the number of senders that are active simultaneously. If, for example, 7 processors are simultaneously sending a message to 10 processors, the average time per multicast is 4.6 msec. This high performance is due to the fact that our protocol is optimized for the common case (i.e., no lost messages). During the experiments described below, the number of lost messages was found to be zero.

We have used the Ethernet implementation for developing several parallel applications written in Orca. Some of these are small, but others are larger. The largest application we currently have is a parallel chess program, consisting of about 2500 lines of code. Smaller applications include matrix multiplication, prime number generation, sorting, and successive overrelaxation. In this section we give preliminary performance measurements of three sample programs running on the Ethernet implementation.

An implementation of Orca designed for a shared-memory multiprocessor would be simpler and, in general, faster than a distributed implementation, since it could put shared objects in the shared memory. Systems with physical shared memory, however, are much harder to build than memory-disjoint systems, especially if a large number of processors (e.g., thousands) is required. To build highly parallel shared-memory systems, a switching network is required, which may be very costly [Almasi and Gottlieb 1989]. It is interesting to compare the performance of our model on distributed and shared-memory systems, and see how much performance is lost by using simpler and less expensive hardware. For this purpose, we also wrote a shared-memory implementation of Orca. This implementation runs on the VME-based multiprocessor described above.

Below, we will compare the performances of the distributed and nondistributed implementations. Both implementations use exactly the same processor boards. The distributed implementation uses the Ethernet for point-to-point and broadcast communication. The nondistributed implementation uses the shared memory for storing shared objects.

4.1. Parallel branch-and-bound

The first application we will discuss is parallel branch-and-bound. As a representative example, consider the traveling salesman problem (TSP). A salesman is given an initial city in which to start, and a list of cities to visit. Each city must be visited once and only once. The objective is to find the shortest path that visits all the cities.

The algorithm we have implemented in Orca uses one *manager* process to generate initial paths for the salesman, starting at the initial city but visiting only part of the other cities. A number of *worker* processes further expand these initial paths, using the "nearest-city-first" heuristic. A worker systematically generates all paths starting with a given initial path and checks if they are better than the current shortest full path. The length of the current best path is stored in a data-object of type *IntObject* (see Figure 1). This object is shared among all worker processes. The manager and worker processes communicate through a shared queue data structure, as shown in Figure 5.

Every time a worker finds a shorter full path, it updates this variable, using the (indivisible) operation *Min*. On the other hand, if a worker ever finds a partial path that is longer than the current best path, it is pointless to continue, so the path being investigated is abandoned.

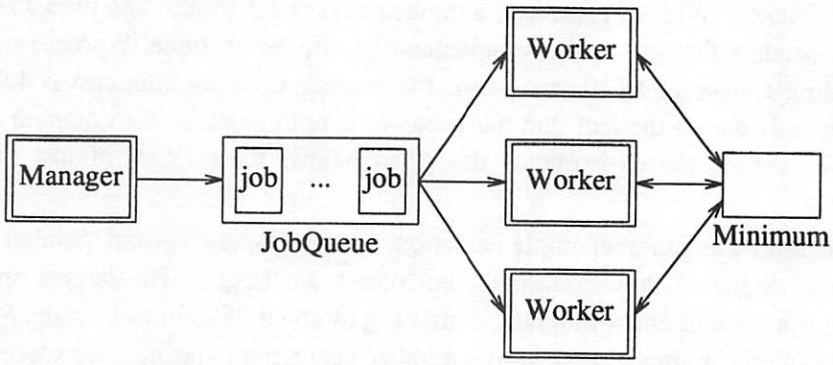


Fig. 5. Structure of the Orca implementation of TSP. The Manager and Workers are processes. The JobQueue is a data-object shared among all these processes. Minimum is a data-object of type IntObject; it is read and written by all workers.

It should be clear that reading of the current best path length will be done very often, but since this is a local operation, there is no communication overhead. Updating the best path happens much less often, but still only requires two broadcast messages (one update message and one acknowledgement).

Although updates of the best path happen infrequently, it is very important to broadcast any improvements immediately. If a worker uses an old (i.e., inferior) value of the best path, it will investigate paths that could have been pruned if the new value had been known. In other words, the worker will search more nodes than necessary. This *search overhead* may easily become a dominating factor and cause a severe performance degradation.

In the RPC model, it is very difficult to let processes share data that are always kept up-to-date. A halfway solution is to let each worker maintain its own local minimum and update this local variable whenever the worker gets a new job. This approach still suffers from a significant search overhead, however [Bal et al. 1987]. With the shared data-object model, on the other hand, sharing data is easy.

The performance of the traveling salesman program (for a randomly generated graph with 12 cities) on the shared-memory and distributed implementations of Orca are given in Figure 6.

With fewer than 5 processors, the shared-memory implementation is slightly faster. This performance difference is caused by the relatively high computational overhead of operations in our prototype distributed implementation. With 6 or more processors, however, the distributed system is faster. (Note that Figure 6 shows the performance for one specific TSP graph; for other randomly generated graphs we have observed similar behavior.)

Although surprising at first sight, this behavior is easy to explain. In the distributed RTS, each processor will have its own local copy of the shared object *Minimum*. Thus, all processors can simultaneously read their copies. In the shared-memory RTS, on the other hand, the object is put in the shared memory and protected by locks, so it becomes a sequential bottleneck.

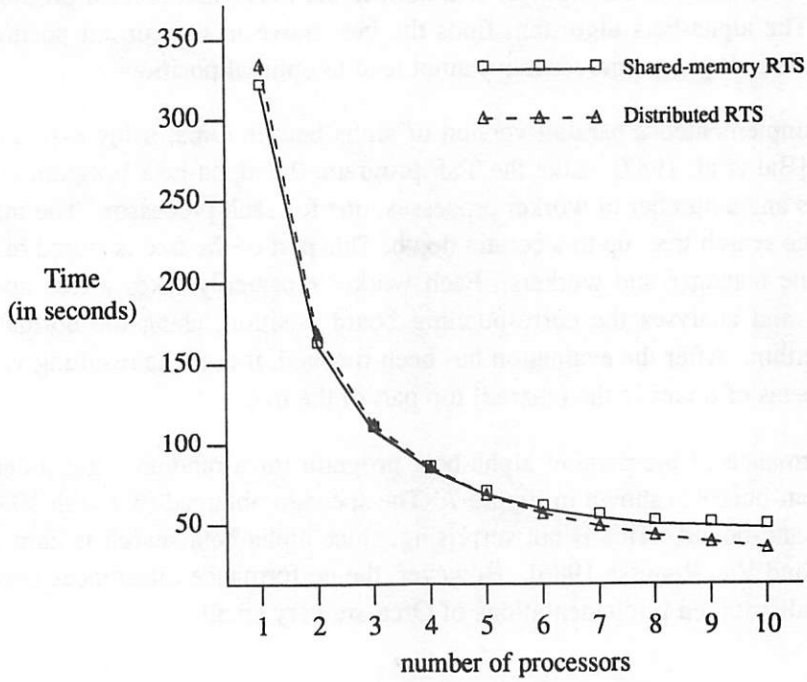


Fig. 6. Measured execution times for the distributed and shared-memory implementations of the Traveling Salesman Problem.

In our prototype implementation of the RTS, the situation is particularly bad, because:

1. Operations are implemented inefficiently and thus are expensive. The *Value* operation, which is used to read the current value of *Minimum*, takes about 40 μsec.
2. Exclusive locks—rather than readers/writer locking—are used.
3. The hardware we use allows only one processor at a time to access the shared memory.

As the *Value* operation is executed very frequently, it will often have to wait for the lock to be free. Undoubtedly, the contention problem would be less severe in a well-tuned shared-memory implementation on more advanced hardware. Still, it is not clear whether the problem can be eliminated entirely in this way, without using local copies of objects.

The distributed implementation achieves almost perfect speedup. With 10 CPUs it is 9.52 times faster than with 1 CPU. The shared-memory implementation achieves a speedup of only 6.71. For comparison, the RPC-based implementation of TSP described in [Bal et al. 1987] achieves a speedup of only 6.29 for the same input graph, using the same hardware. The lower speedup of the RPC implementation is caused by its high search overhead.

4.2. Parallel alpha-beta search

Alpha-beta search is an efficient method for searching game trees for two-person, zero-sum games (e.g., chess). A node in such a game tree corresponds to a position in the game. Each node has one branch for every possible move in that position. A value associated with the node indicates how good that position is for the player who is about to move. At even levels of the

tree, this value is the *maximum* of the values of its children; at odd levels it is the *minimum*, as the search algorithm assumes each player will choose the move that is least profitable for his or her opponent. The alpha-beta algorithm finds the best move in the current position, searching only part of the tree. It prunes moves that cannot lead to optimal positions.

We have implemented a parallel version of alpha-beta in Orca, using essentially the same algorithm as in [Bal et al. 1987]. Like the TSP program, the alpha-beta program consists of one manager process and a number of worker processes, one for each processor. The manager builds the top part of the search tree, up to a certain depth. This part of the tree is stored in a data-object shared among the manager and workers. Each worker repeatedly takes a leaf node of the top part of the tree and analyses the corresponding board position, using the normal (sequential) alpha-beta algorithm. After the evaluation has been finished, it uses the resulting value to update the alphas and betas of nodes in the (shared) top part of the tree.

The performance of the parallel alpha-beta program for a randomly generated search tree of depth 6 and fan-out 38 is shown in Figure 7. The speedup obtained (6.4 with 10 CPUs) is less than for branch-and-bound. This is not surprising, since alpha-beta search is hard to parallelize efficiently [Bal and Van Renesse 1986]. However, the performance differences between the distributed and nondistributed implementations of Orca are very small.

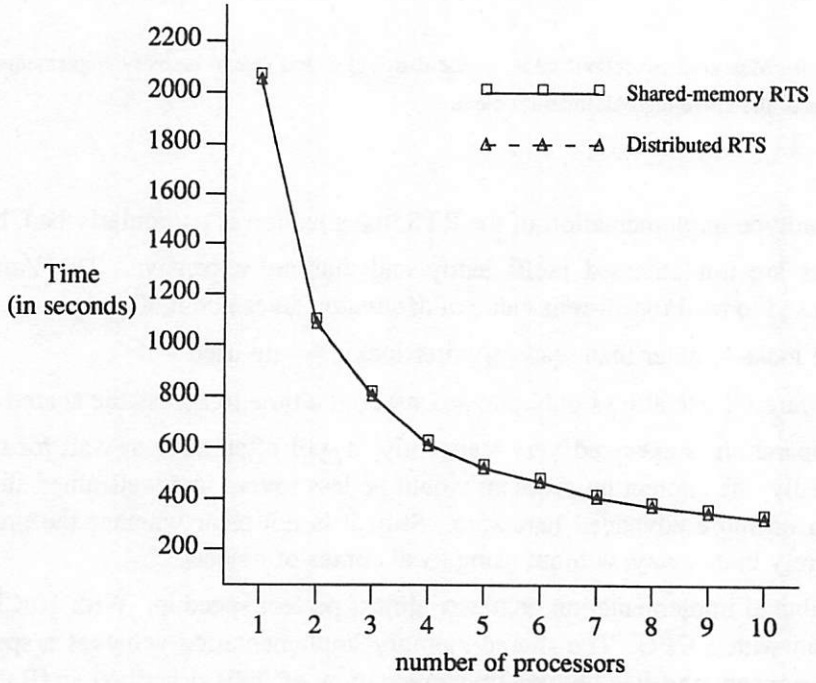


Fig. 7. Measured execution times for the distributed and shared-memory implementations of Alpha-Beta search.

4.3. Parallel all-pairs shortest paths problem

The third and last application we describe here is the All-pairs Shortest Paths problem. In this problem it is desired to find the length of the shortest path from any node *i* to any other node *j* in a given graph. The parallel algorithm we use is similar to the one given in [Jenq and Sahni 1987], which is a parallel version of Floyd's algorithm. The distances between the nodes are represented in a matrix. Each processor computes part of the result matrix. The algorithm requires a nontrivial amount of communication and synchronization among the processors.

The performance of the program (for a graph with 200 nodes) on our two implementations is given in Figure 8. The shared-memory implementation is slightly more efficient. The performance difference is caused by the high communication overhead of the algorithm. The parallel algorithm performs 200 iterations; after each iteration, an array of 200 integers is sent from one processor to all other processors. In spite of this high communication overhead, the distributed implementation still has a good performance. With 10 CPUs, it achieves a speedup of 9.17 (as opposed to 9.48 for the shared-memory system). One of the main reasons for this good performance is the use of broadcast messages for transferring the array to all processors.

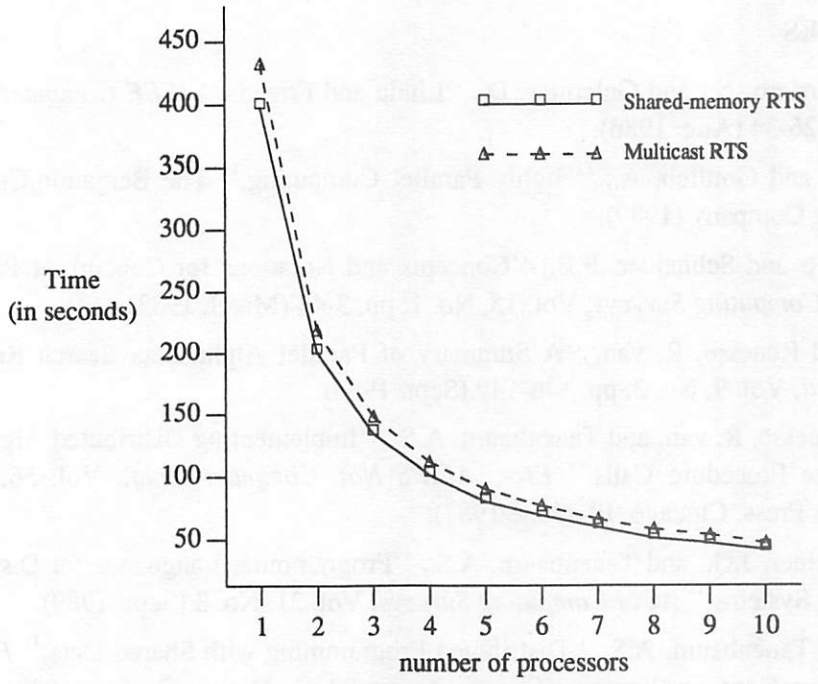


Fig. 8. Measured execution times for the distributed and shared-memory implementations of the All-pairs Shortest Paths problem.

5. CONCLUSION

We have described a new model and programming language for implementing parallel applications on distributed systems. In contrast with most other models for distributed programming (e.g., the RPC model), our model allows processes on different machines to share data. The implementation of the model takes care of the physical distribution of shared data among

processors. In particular, the implementation replicates shared data, so each process can directly read the local copy on its own processor.

The main purpose of this paper was to show that, for several applications, our model is both easy to use and efficient. We have studied one distributed implementation of our language and measured the performance of three applications. Our model is best suited for moderate-grained parallel applications in which processes share data that are read frequently and modified infrequently. As a good example, the TSP program of Section 4.1 uses a global variable that is read very frequently and is changed only a few times. This program shows an excellent performance. In the two other applications (Alpha-Beta search and the All-pairs Shortest Paths problem), the shared data are changed more frequently. Still, the performances of these applications are high, because we use an efficient mechanism for updating replicas, based on broadcasting rather than point-to-point messages.

ACKNOWLEDGEMENTS

We would like to thank Wim van Leersum for implementing the Orca compiler and Erik Baalbergen, Arnold Geels, and the anonymous referees for giving useful comments on the paper.

REFERENCES

Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *IEEE Computer*, Vol. 19, No. 8, pp. 26-34 (Aug. 1986).

Almasi, G.S. and Gottlieb, A., "Highly Parallel Computing," The Benjamin/Cummings Publishing Company (1989).

Andrews, G.R. and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, No. 1, pp. 3-43 (March 1983).

Bal, H.E. and Renesse, R. van, "A Summary of Parallel Alpha-Beta Search Results," *ICCA Journal*, Vol. 9, No. 3, pp. 146-149 (Sept. 1986).

Bal, H.E., Renesse, R. van, and Tanenbaum, A.S., "Implementing Distributed Algorithms Using Remote Procedure Calls," *Proc. AFIPS Nat. Computer Conf.*, Vol. 56, pp. 499-506, AFIPS Press, Chicago, Ill. (June 1987).

Bal, H.E., Steiner, J.G., and Tanenbaum, A.S., "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3 (Sept. 1989).

Bal, H.E. and Tanenbaum, A.S., "Distributed Programming with Shared Data," *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, pp. 82-91, Miami, Fl. (Oct. 1988).

Birman, K.P. and Joseph, T.A., "Reliable Communication in the Presence of Failures," *ACM Trans. Comp. Syst.*, Vol. 5, No. 1, pp. 47-76 (Feb. 1987).

Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59 (Feb. 1984).

Bisiani, R. and Forin, A., "Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems," *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30, Palo Alto, Calif. (Oct. 1987).

Chang, J. and Maxemchuk, N.F., "Reliable Broadcast Protocols," *ACM Trans. Comp. Syst.*, Vol. 2, No. 3, pp. 251-273 (Aug. 1984).

Cheriton, D.R., "Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems," *ACM Operating Systems Review*, Vol. 19, No. 4, pp. 26-33 (Oct. 1985).

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM*, Vol. 19, No. 11, pp. 624-633 (Nov. 1976).

Garcia-Molina, H. and Spauster, A., "Message Ordering in a Multicast Environment," *Proc. 9th Int. Conf. on Distr. Comp. Syst.*, pp. 354-361, Newport Beach, CA (June 1989).

Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Commun. ACM*, Vol. 17, No. 10, pp. 549-557 (Oct. 1974).

Jenq, J.-F. and Sahni, S., "All Pairs Shortest Paths on a Hypercube Multiprocessor," *Proc. of the 1987 Int. Conf. on Parallel Processing*, pp. 713-716, St. Charles, Ill. (Aug. 1987).

Joseph, T.A. and Birman, K.P., "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 4, No. 1 (Feb. 1987).

Jul, E., Levy, H., Hutchinson, N., and Black, A., "Fine-Grained Mobility in the Emerald System," *ACM Trans. Comp. Syst.*, Vol. 6, No. 1, pp. 109-133 (Feb. 1988).

Kaashoek, M.F., Bal, H.E., and Tanenbaum, A.S., "Experience with the Distributed Data Structure Paradigm in Linda," *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL. (Oct. 1989a).

Kaashoek, M.F., Tanenbaum, A.S., Flynn Hummel, S., and Bal, H.E., "An Efficient Reliable Broadcast Protocol," Report IR-195, Vrije Universiteit, Amsterdam, The Netherlands (July 1989b).

Li, K., "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. 1988 Int. Conf. Parallel Processing (Vol. II)*, pp. 94-101, St. Charles, Ill. (Aug. 1988).

Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Commun. ACM*, Vol. 19, No. 7, pp. 395-404 (July 1976).

Mullender, S.J. and Tanenbaum, A.S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, Vol. 29, No. 4, pp. 289-300 (March 1986).

Renesse, R. van, Staveren, J.M. van, and Tanenbaum, A.S., "The Performance of the Amoeba Distributed Operating System," *Software—Practice and Experience*, Vol. 19, No. 3, pp. 223-234 (March 1989).

Tanenbaum, A.S. and Renesse, R. van, "Distributed Operating Systems," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470 (Dec. 1985).

Tanenbaum, A.S. and Renesse, R. van, "A Critique of the Remote Procedure Call Paradigm," *Proc. of the EUTECO 88 Conf.*, pp. 775-783, ed. R. Speth, North-Holland, Vienna, Austria (April 1988).

Tanenbaum, A.S., Staveren, H. van, Keizer, E.G., and Stevenson, J.W., "A Practical Toolkit for Making Portable Compilers," *Commun. ACM*, Vol. 26, No. 9, pp. 654-660 (Sept. 1983).

An Implementation of Distributed Shared Memory*

Umakishore Ramachandran
School of ICS
Georgia Institute of Technology
Atlanta, Ga 30332
Phone: (404) 894-5136
E-mail: rama@gatech.edu

M. Yousef A. Khalidi
Sun Microsystems
2550 Garcia Avenue
Mountain View, Ca 94043
Phone: (415) 336-5392
E-mail: yak@sun.com

Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global *distributed shared memory*. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

1 Introduction

Programming with shared memory is well-understood and despite the interest in distributed and parallel systems for reasons of availability, fault-tolerance, and increased computational power, the style of programming these systems has not changed drastically. Even in non-shared memory architectures researchers have proposed a style that presents to the programmers an abstraction of a logical shared memory [19, 14, 8, 23]. Other researchers have proposed algorithms for maintaining the consistency of such a logically shared memory in non-shared memory architectures [17, 18, 21]. The abstraction for supporting the notion of shared memory on a non-shared memory (distributed) architecture is referred to as *distributed shared memory* (DSM) in this paper.

A second motivation for DSM is the current trend in structuring distributed systems using a collection of diskless computational servers, namely workstations, and a few data servers or file servers. In such an environment the code and data for program execution has to be paged-in from the data server. There are two issues here: The first one is a scheduling decision of 'where' to execute the program, one that is best left to a higher level policy making entity. The second one is the chore of bringing in the required data and code, i.e., remote paging. If sharing is coupled with this second issue, then we see that DSM presents itself as a natural facility for combining the two.

Several other researchers have proposed software architectures based on the shared memory paradigm, in different settings:

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809268.

- Li [18] presents a variation of the Berkeley protocol for multiprocessor cache consistency [15] as a solution to maintain the consistency of distributed shared memory. Using Li's scheme, the entire memory in the distributed system is considered potentially sharable for both reads and writes. The consistency protocol maintains the coherency of memory even when accessed by processes running on different nodes.
- In a speech recognition application, Bisiani and Forin [8] use data structures that are shared by multiple language modules that are distributed on heterogeneous machines. They show that communication through shared memory is a viable alternative to message-passing even when the environment involves cooperation between multilingual program modules and heterogeneous machines.
- Processes in the programming language *Linda* [9, 13] communicate via a globally-shared collection of ordered tuples.
- A logically shared bulletin-board is proposed by Birman, et al. [7] for structuring asynchronous interactions between processes in distributed systems.
- By integrating the mechanisms for virtual memory management and local interprocess communication, Mach [25] achieves efficient implementation of local interprocess communication. Currently, researchers at CMU are investigating the duality of shared memory and message passing in the context of network communication as well [30].
- Zayas [31] achieves substantial reduction in the cost of process migration by using copy-on-write techniques [24] and on-demand fetches during remote execution.
- Cheriton [10] advocates problem-oriented shared memory as the basic concept for structuring distributed systems.
- Emerald [14] is a distributed object-based language and system with support for object mobility.

The purpose of this paper is to present a set of mechanisms for DSM and an implementation of these mechanisms. All the resources of the system are viewed as potentially shared objects. The name space of these objects constitute a distributed shared memory. The objects are composed of segments, where a segment is a logical entity that has attributes such as read-only, and read-write. There is a concept of *ownership* and the node where a segment is created (the owner node) is responsible for guaranteeing the consistency of the segment. The *distributed shared memory controller* (DSMC) to be described next is the entity that provides the mechanisms for managing these segments.

2 Distributed Shared Memory Controller

The basic operations provided by the DSMC are `get`, and `discard`. The `get` operation is used to fetch a segment from its owner, while `discard` is used to return a segment to its owner. The DSMC provides synchronization primitives as separate operations (P and V semaphore operations), or as combined access and lock operations using the `get` and `discard` primitives.

Using the `get` primitive a segment may be acquired in one of four modes: `read-only`, `read-write`, `weak-read`, or `none`. `Read-only` mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. `Read-write` mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. When a `get`

primitive is issued with mode `read-only` or `read-write` the local DSMC sends a request to the owner DSMC and suspends the requesting process until the segment is received. The segment is kept until an explicit `discard` is issued. Multiple copies of the segment may be held by several readers at the same time (mode `read-only`) but only one writer (mode `read-write`) may have access to the segment at a time. The owner node keeps a count of the number of requesters that have a copy of the segment in `read-only` mode.

`Weak-read` mode signifies non-exclusive access with no guarantee whether the segment will change or not. The owner DSMC immediately honors a `weak-read` requests by sending a copy of the segment to the requesting DSMC. `None` mode signifies exclusive access with no guarantee whether the segment will be thrown away or not. When a `get` primitive is issued with mode `none` the local DSMC sends a request to the owner DSMC and suspends the requesting process until the segment is received. `None` mode requests are enqueued in the appropriate segment queue, if the segment is currently held in either `read-only` or `read-write` modes. If the segment is available at the owner DSMC, it responds by sending the segment to the requesting DSMC. The requesting DSMC becomes the *keeper* of the segment and the owner remembers the current keeper. If the segment is held in another node in mode `none`, then the owner DSMC instructs the current keeper to forward the segment to the requesting DSMC. A segment held in mode `none` at a keeper node may be returned to its owner by issuing a `discard` primitive, or it may be taken away by its owner when the keeper DSMC is instructed to forward the segment to another node.

The DSMC also provides the semaphore operations P and V that act on semaphores that are contained in semaphore segments (see §5).

3 Clouds

While the mechanisms provided by DSMC are general, we describe an implementation of these mechanisms in the context of *Clouds*, an object-based distributed operating system. Therefore, a brief description of Clouds is appropriate.

Clouds, being developed at Georgia Tech [5], is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of Clouds. *Objects* and *threads* are the basic building blocks of Clouds. Objects are passive entities and specify a *distinct* and *disjoint* piece of the global virtual address space that spans the entire network. An object is the encapsulation of the *code* and *data* needed to implement the *entry points* in the object. Thus a Clouds object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in software.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. Figure 1 shows the model of computation in Clouds. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes.

In a distributed object-based system, the virtual address spaces of all objects can be viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-management mechanism.

For remote object invocation there are two choices: The first choice is to perform the computation at the node where the object resides, referred to as remote procedure call. The

second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects, such as an object that reads disk blocks, we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [12] that suggests an invocation or other invocations in the same object may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [16, 22].

4 The Structure of Clouds

4.1 Ra Kernel

Ra [4, 6] is an operating system kernel designed to be the nucleus of Clouds operating system [5]. It is currently implemented on the Sun-3 architecture. Ra defines and manages three primitive abstractions: *segment*, *virtual space*, and *isiba*. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is *attached* to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a *partition* that is responsible for providing backing store for the segment. A partition is an entity that realizes, maintains, and manipulates segments (see §4.2).

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is a monotonically increasing range of virtual addresses with possible "holes" in the range. A virtual space has a descriptor segment associated with it that contains a collection of windows. Each window is a data structure that maps a contiguous piece of the virtual space to a segment. Figure 2 shows the relationship between the Ra virtual space, the windows, and the segments. The segmentation scheme in the Chorus system [2] has some similarity to the Ra virtual space.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as light-weight processes. Isibas may be used as daemons within the kernel or they may be associated with a Ra virtual space to implement a user process. A Clouds thread can potentially span machine boundaries and is implemented as a collection of processes.

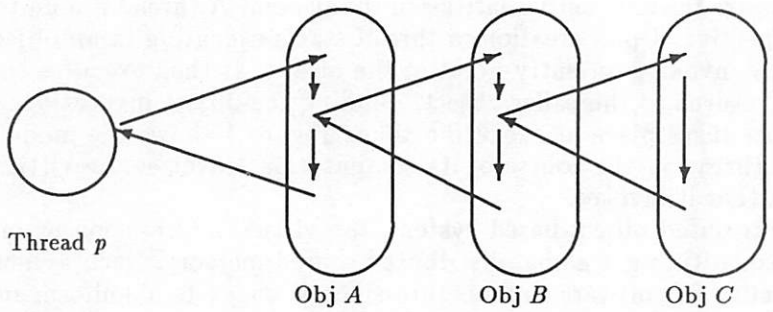


Figure 1: Model of Computation

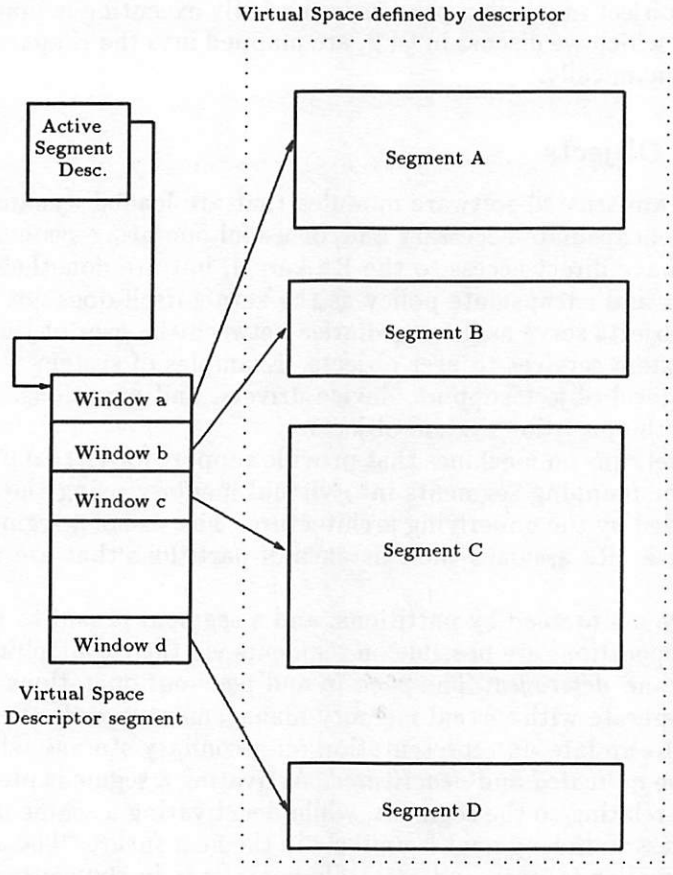


Figure 2: Ra Virtual Space

A Ra virtual space is a software abstraction not to be confused with the virtual address space provided by the machine architecture. The latter is assumed to be composed of three distinct regions that are called O, P, and K spaces for object, process, and kernel, respectively. Note that such a distinction may not exist in a given machine architecture. In that case the division is enforced based on address range 'high' and 'low' water marks.

The kernel is mapped in the K space. A process consists of an isiba and a Ra virtual space that contains the process stack for invocations. Note that a process' virtual space does not contain any code. A process' virtual space is mapped into the P space and unmapped on context switch. An object is a Ra virtual space that consists of code and data segments. The code segment of an object's virtual space has entry points that can be invoked by user processes. The object in which a process is currently executing is mapped into the O space. System objects, which we discuss in §4.2, are mapped into the K space, but may be installed and removed dynamically.

4.2 System Objects

System objects are trusted software modules that are loaded dynamically in the K space. System objects encapsulate necessary and/or useful operating system services and resource managers that have direct access to the Ra kernel, but are nonetheless outside the kernel. They implement and encapsulate policy as the kernel itself does not make any policy decisions. System objects serve as intermediaries between the user objects and the kernel, and they provide system services to user objects. Examples of system objects include resource managers, user-level object support, device drivers, and partitions. Of particular concern to this paper is the partition system objects.

The Ra kernel runs on machines that provide support for virtual memory. The Ra kernel is responsible for mapping segments into virtual memory using the memory management hardware provided by the underlying architecture. The size of a segment is a multiple of the physical page size. Ra assumes the existence of partitions that are responsible for storing segments.

Segments are maintained by partitions, and a segment is said to be controlled by a partition. Several operations are possible on segments via their controlling partition: Segments may be *created* and *destroyed*. The *page-in* and *page-out* operations on segments allow the partition to cooperate with virtual memory management in order to access the contents of a segment and to update its representation on secondary storage when necessary. Finally, segments may be *activated* and *deactivated*. Activating a segment prepares the partition for further activity relating to the segment, while deactivating a segment informs the partition that further access to the segment is unlikely in the near future. The activate and deactivate operations are similar to *open* and *close* file operations in conventional systems.

Therefore, each partition provides, at least, the following calls for use by Ra: activate/deactivate segment, create/destroy segment, and page-in/page-out portions of segments. Ra services segment requests from other system objects, such as to map a segment into a virtual space. In addition, Ra fields page faults, determines the virtual space and in turn the segment where the fault occurred, and calls the appropriate fault handler. When Ra is instructed to service a segment request (e.g. to map a segment into a virtual space), it invokes the appropriate partition to fetch the segment into physical memory. Ra then manipulates the memory management hardware to map the physical pages appropriately.

The Ra architecture assigns to the kernel the task of mapping a segment onto the memory management hardware, and hides the details of storing the segment on external storage in the partition system objects. The fact that the segment is stored on a local disk or on a remote node is hidden from the kernel.

5 Implementation on Ra

5.1 Overview

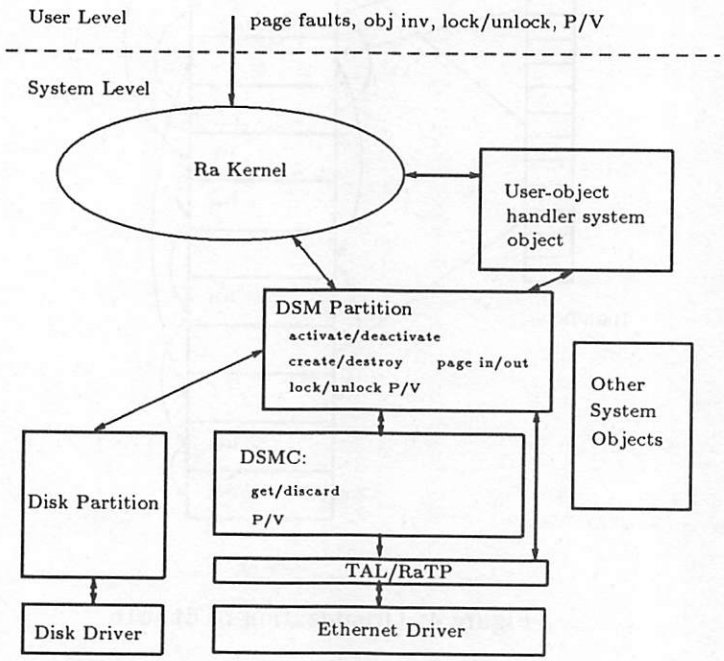


Figure 3: Organization of DSMC implementation under Ra

Figure 3 shows the organization of the DSMC implementation (roughly 3500 lines of C++ code [26]) on Ra. The boxes in the Figure denote system objects. The DSMC cooperates with remote DSMC's to implement the distributed shared memory primitives. The DSM partition is a Ra partition that provides the kernel with the ability to create/destroy and activate/deactivate segments, page-in/page-out portions of segments, and semaphore P/V operations. The DSM partition decides if a segment is owned by the local node or a remote node. It uses the Disk partition to access local segments, and uses the DSMC to access remote segments. The Disk partition maintains segments owned by the local node on the local stable storage (if any).

The DSMC algorithms require simple reliable request/response messages (possibly with message forwarding). In our implementation, we use the *transaction abstract layer* that is built on the *Ra Transaction Support Protocol* (TAL/RaTP) [29]. TAL/RaTP protocol is similar to other transaction-oriented protocols such as VMTP [11]. However, it is much simpler than VMTP since it is tailored to our requirements.

5.2 Handling Local Requests

5.2.1 DSM Partition

DSM partition provides the minimum set of partition operations plus the semaphore operations. The DSM partition handles segment requests from the Ra kernel or from other system objects (Figure 3). An example of a system object that uses the DSM partition is the user-level object handler that is responsible for implementing object invocation and servic-

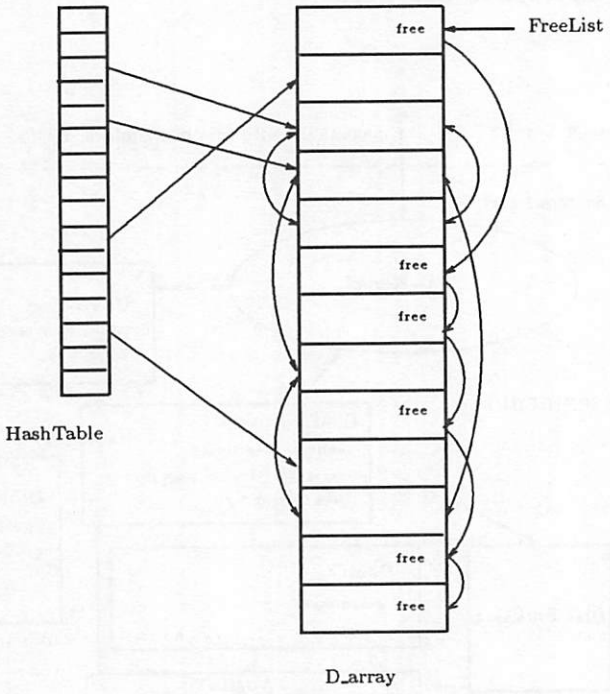


Figure 4: Organization of `dtable`

ing user-visible segment operations. Such user-visible operations may include `lock/unlock` and `P/V` operations.

The DSM partition maintains the status of cached (local and remote) segments on the local node in a table called `dtable` (Figure 4). Each `dtable` entry maintains information about a block of a segment (where a block is a multiple of the physical page size). In the current implementation, a block is equal to the physical page size on the Sun-3 (8K bytes). Each valid entry in the `dtable` is doubly linked on a hash list. The table is hashed by the segment name, and searched with the key <segment name, block#>. All free entries are linked on a free list. In addition, an *active segment table* (`ast`) contains an entry per active segment on the local node.

Each `dtable` entry represents one segment block and includes the following fields:

- `segment`, `block_number` — These two fields identify the segment block represented by this entry.
- `wait_lock` — A lock that is used to synchronize access to this entry.
- `phys_frame` — An array of physical frame numbers that contain this block (in the current implementation, the cardinality of the array is one).
- `pending` — A flag indicating that a read from disk is in progress, or a `get` with mode `none` has been issued to the owner DSMC.
- `Mode` — This field indicates the current mode of the block.
- `readers` — A field that indicates the number of requesters that have this block in `read-only` mode.

- `owner_flag` — A flag that indicates if the local node is the owner of the segment.
- `keeper_owner` — This field gives the current keeper of the segment (if `owner_flag` is true), or the owner of the segment (if `owner_flag` is false).
- `squeue` — A list used by the owner DSMC to queue `read-only` and `read-write` requests for this block from remote DSMCs.

The DSM partition operations can be classified into three groups:

- *Control Operations*:

  `activate`(segment)
  `deactivate`(segment)
  `create`(segment)
  `destroy`(segment)

  The control operations search the `ast` for an entry describing the segment. If an entry is not found, the *location system object* is consulted for the location of the segment.[1] If the segment is available on a local disk partition, the corresponding control operation is invoked on the disk partition. Otherwise, the segment is owned by a remote node, and a `msg_control` message is sent to the DSM partition on the remote node. Note that locating the segment is the responsibility of the location system object, and that the DSMC is not involved in handling any of the control operations.

- *Data Transfer Operations*:

  `page_in`(segment, block, physical page)
  `page_in`(segment, block, physical page, mode)
  `page_out`(segment, block)

  The `page_in` operation activates the segment, if necessary. The `page_in` operation searches the `dtable` for an entry describing <segment, block>. If no such entry is found, an entry is created. For segments owned by the local node, the `page_in` operation on the disk partition is invoked. For remote segments, the DSM partition translates the `page_in` requests to the DSMC `get` operations with the specified mode. If no mode is indicated in the `page_in` call, mode `none` is assumed. The `page_out` operations locates the `dtable` entry describing <segment, block>, and invokes the `page_out` call on the disk partition if the segment is local, or calls the DSMC `discard` operation, if the segment is remote.

- *Synchronization Operations*:

  `P`(segment, semaphore_num)
  `V`(segment, semaphore_num)

  Semaphores are stored in *semaphore segments*. Semaphore segments have the format shown in Figure 5. Each semaphore segment consists of three parts: a `descriptor` structure, n `semaphore` structures, and m `block` structures. `Descriptor` contains the number of `semaphore` and `block` structures, a bitmap of free/used `semaphore`

[1] Given the name of a segment, the location system object returns the location of the segment owner. A simple location system object broadcasts a search request for each location operation (see [3, 1] for more sophisticated location algorithms).

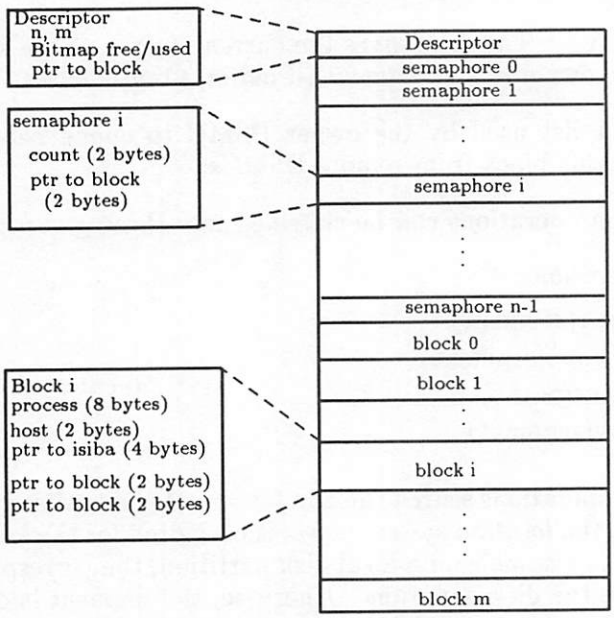


Figure 5: Structure of a semaphore segment

structures, and a pointer to a free list of `block` structures. Each `semaphore` structure describes a semaphore, and includes a counter and a pointer to a doubly-linked list of `block` structures. Each `block` structure describes a process waiting for a semaphore. Each `block` contains the name of the waiting process, the host name where the process is blocked, and a pointer to the *isiba control block* (ICB) on the host where the process is blocked. Note that the host name and ICB pointers are *hints* to the location of the blocked process because processes can migrate from one node to another. The process name field is an absolute pointer to the process and can be used by the locator system object to find the process if necessary. Processes (as well as other entities in Ra) have unique network-wide names.

Semaphore segments can be attached to a range of virtual addresses like any other segment in Ra. Therefore, they can be initialized and manipulated directly. The DSM partition maintains a table (called `semtable`) that describes each semaphore in use. `Semtable` acts as a cache of the active semaphores that are in the semaphore segments. `Semtable` is organized in a similar fashion as the `dtable`, but it is searched using the key <segment, semaphore number>. Each `semtable` entry caches exactly one semaphore from a semaphore segment. The structure of each `semtable` entry is the same as the `semaphore` structure in semaphore segments (see Figure 5). A pool of in-memory `block` structures are used to cache contents of `block` structures from semaphore segments.

Operations on a semaphore that belongs to a local segment are performed locally by reading the segment from the disk partition, initializing a `semtable` entry, and then performing the operations on the `semtable` entry. In the current implementation, all semaphore operations are performed at the node that owns the semaphore segment. Therefore, operations on semaphores that belong to remote segments are translated into DSMC `P` and `V` operations (see §5.5).

Type	From	To	Description
msg_control	DSM part.	DSM part.	used for control operations
msg_reply	DSM part.	DSM part.	reply to msg_control
msg_get	keeper DSMC	owner DSMC	fetches a segment block
msg_discard	keeper DSMC	owner DSMC	returns a segment block
msg_forward	owner DSMC	keeper DSMC	forwrds a segment block
msg_segment	any DSMC	any DSMC	delivers requested segment block
msg_P	keeper DSMC	owner DSMC	semaphore P operation
msg_V	keeper DSMC	owner DSMC	semaphore V operation
msg_unblock	owner DSMC	keeper DSMC	continues suspended process
msg_error	any	any	indicates an exception

Table 1: Summary of TAL messages

5.2.2 DSMC

The DSMC provides the following four operations for use by the DSM partition:

get(dtable_index, mode)

discard(dtable_index)

P(semtable_index)

V(semtable_index)

In order to implement these primitives, each DSMC uses the TAL/RaTP messages listed in Table 1 to communicate with other DSMCs (see §5.3.2).

5.3 Handling Remote Requests

5.3.1 DSM Partition

DSM partitions exchange msg_control and msg_reply messages to implement the activate, deactivate, create, and destroy control operations, as described in §5.2.1. In addition, each DSM partition services requests from its local DSMC to activate local segments, to read a block of a local segment, and to initialize a semtable entry from a local semaphore segment (see §5.3.2).

5.3.2 DSMC

The DSMC may receive several messages from remote DSMCs. We describe the DSMC action for each message received:

- msg_get(segment, block, mode)
  If there is no ast entry for this segment, the DSMC asks the DSM partition to activate the segment and to read the required block into memory. The DSMC examines the dtable entry describing the required block. Depending on the information contained in the dtable, the DSMC performs one of the following actions:
  - If the segment mode does not conflict with the requested mode and the block is available locally, send a msg_segment message that includes the block to the requesting DSMC

- If the segment mode is `none`, the requested mode is also `none` and the segment is held at a remote node, then send a `msg_forward` message to the remote DSMC, instructing it to forward the block to the requester
- If the segment mode and requested mode conflict (e.g. segment is held in `read-only` mode and requested mode is `read-write`), queue request until segment is available
- If the segment does not exist at this node, then send a `msg_error` message to the requesting DSMC.

- `msg_discard`(segment, block)
  The DSMC updates the `dtable` entry describing the discarded block. If there exist any pending `get` requests for this block that now can be satisfied, they are serviced by sending `msg_segment` messages to the requesting DSMCs.

- `msg_segment`(segment, block)
  The DSMC receives a `msg_segment` message as a response for a `msg_get` message. The DSMC locates the `dtable` entry describing the block, and resumes the suspended processes that are awaiting the arrival of the block.

- `msg_forward`(segment, block, destination host)
  The DSMC informs the DSM partition that this block is no longer available, and then issues a `msg_segment` message containing the required block to the destination host.

- `msg_P`(segment, semaphore_num)
  If there is no `semtable` entry for the required semaphore, the DSMC requests its local DSM partition to initialize an entry. The DSM partition may have to activate the required semaphore segment and then read the information of semaphore `semaphore_num` into the new `semtable` entry. The DSMC decrements the semaphore count, and responds with a `msg_unblock` message if the count is greater than or equal to zero. Otherwise, it links to the `semtable` entry a new `block` structure that describes the requesting process.

- `msg_V`(segment, semaphore_num)
  If there is no `semtable` entry for the required semaphore, the DSMC requests its local DSM partition to initialize the entry. The DSMC increments the semaphore count. If the count is less than or equal to zero, the DSMC unlinks the first `block` structure from the `semtable` entry and sends a `msg_unblock` to resume the process described by the unlinked `block` structure.

- `msg_unblock`(segment, semaphore_num, ptr to ICB)
  The DSMC resumes the execution of the waiting process identified by the `msg_unblock` message.

- `msg_error`(segment, block, error_type)
  An error indication may be received if a request cannot be satisfied. The error_type field gives the reason for the failure of the request. Upon receiving an `msg_error` message as a response for a request, the DSMC returns an error indication to the original requester.

5.4 Table Management

As mentioned in §5.2.1, the DSM partition is responsible for maintaining `ast`, `semtable`, and `dtable` structures. The size of `ast` is equal to the maximum number of active segments

at any point of time, and `ast` entries can be reclaimed when the segments they represent are deactivated. `Semtable` acts as a cache of the information contained in semaphore segments, and its size is equal to the expected number of semaphores in use. To reclaim a `semtable` entry, the contents of the entry has to be written back into its semaphore segment. The size of the `dtable` depends on whether or not the node acts as an owner of segments. For nodes that act only as keepers, the size of the `dtable` is less than or equal to the number of physical pages at the node. For a node that acts as an owner, the size of the `dtable` is determined by the number of nodes serviced and the size of their physical memories. A `dtable` entry is reclaimed when the block it represents is paged-out. If the entry represents a block that is cached at a keeper node, the entry can be reclaimed when the block is returned to the owner.

5.5 Performing Semaphore Operations Locally

As described in §5.2.1, all semaphore operations in the current implementation are performed at the node that owns the semaphore segment. To exploit synchronization locality (e.g. when all processes using the same semaphore are at the same node), it should be possible to perform the semaphore operations at the local node without the intervention of the owner node on each operation.

Because semaphores reside in segments, it is possible to fetch the semaphore segment from its owner, read its contents into the local `semtable`, and perform the operations locally. When a decision is made to move the semaphore segment from its current node, the DSM partition must ensure that the contents of the segment is up to date by flushing any `semtable` entries that belong to the segment prior to sending the segment to another node. Processes blocked on a semaphore need not be migrated when the semaphore segment is moved to another node, because the semaphore segment contains the name of the host where the process is blocked (Figure 5).

The following simple modifications to the DSMC `P` and `V` primitives are required: Instead of sending all semaphore operations to the owner DSMC, a check is made to see if the semaphore segment is cached locally. If it is, the operation is performed locally and the owner DSMC is not contacted. Otherwise, the semaphore operation is sent to the owner DSMC. When the owner DSMC receives a semaphore operation message (`msg_P` or `msg_V`), it checks to see whether the required semaphore segment is available locally, or is cached at a remote keeper. If the semaphore segment is available locally, the semaphore operation is performed as before at the owner node. Otherwise, the semaphore operation is forwarded to the current keeper of the semaphore segment. Note that the owner DSMC maintains at all times the location of the current keeper of each segment, and therefore can easily forward semaphore operations on segments that are cached at remote keepers.

The semaphore mechanisms presented in this section do not address the issue of where (or when) to move semaphore segments. Instead, they perform the P and V semaphore operations at the *current* location of the semaphore segment. Other system objects are responsible for deciding on where to place semaphore segments in the distributed system, and when to move them to other nodes.

5.6 Fault Tolerance

The DSMC implementation assumes the existence of a reliable transport protocol underneath. Any failures in the network results in an 'error' indication being propagated to the system object that made the request to the DSMC. Recovering from such failures could possibly involve reconstructing the segments at a different node. Failure handling clearly involves policy issues, best left to appropriate system objects. The DSM layer concerns itself

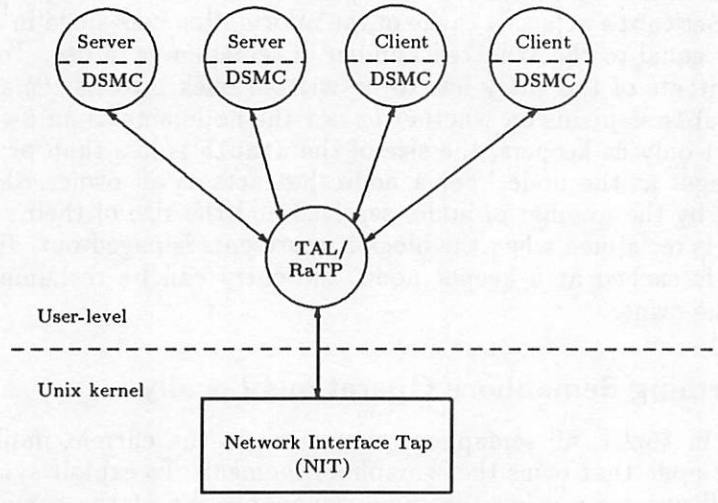


Figure 6: Organization of DSMC implementation on Unix

only with segment transport and gives the necessary error indication to higher level system objects for appropriate corrective action. In our view, fault-tolerance has to be addressed in distributed systems regardless of whether RPC or DSM is used as a mechanism for remote invocation, and we plan to do this as part of our future research.

6 Implementation on Unix

The DSMC and the DSM partition have been implemented on top of Unix as well. This implementation serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL/RaTP protocols.

2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.

3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is the transparent management of distributed data and computation. Providing inter-operability between Unix and Clouds is one of our design goals. DSM implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via the DSM mechanisms.

The organization of the DSMC implementation on Unix is shown in Figure 6. TAL/RaTP runs as a user process that uses SUN's Network Interface Tap (NIT) [27] to receive packets from the net and to route them among a set of clients and servers. The DSMC code is linked-in with the server code that uses the Unix file system to store segments and service requests from Ra DSM partitions. The DSM code is also linked-in with client code that is used to test the DSM system.

Most of the DSMC and DSM partition code that runs on Ra are re-used for the Unix implementation, and the operating system dependencies are isolated in a few C++ classes. To enable more than one Unix process to share the DSM tables, we use Unix System V shared memory regions [27]. In addition, Unix System V semaphores are used to synchronize

access to the tables. In our initial implementation, we also used System V semaphores to synchronize the TAL/RaTP process and its client processes. A client process requested TAL/RaTP services by writing in a shared region of memory and blocking on a semaphore. TAL/RaTP eventually resumed the process by issuing a V operation on the semaphore.

In the current implementation, however, we switched to using Unix 4.2 BSD socket IPC primitives instead of System V semaphores because of the poor performance of the initial implementation (see §7). The TAL/RaTP process communicates with its client processes through shared memory and sockets. Processes communicate their requests to TAL/RaTP via socket IPC primitives. However, data blocks are passed through shared memory regions to minimize copying.

7 Performance Evaluation

In this section, we report on the performance of the DSM implementation on Unix and Ra. All measurements are done on Sun-3/60 workstations with 4M bytes of memory, connected through a 10M bits/sec ethernet. We mask out the cost of secondary storage access by caching segments in memory before measuring the costs of the DSMC primitives.

7.1 Unix

The implementation on Unix is complete and Table 2 summarizes the results. The Table shows that on an average fetching a segment (without forwarding) of size 8K bytes (the page size on the Sun-3) takes 43.4 ms. Van Renesse et al. report a transfer rate of 40 ms for 8K bytes between two user processes on different nodes using Sun RPC on a 10M bits/sec ethernet [28]. Our implementation uses two user processes per node and still compares favorably with the figures reported by van Renesse et al. A null message from one DSMC to another costs roughly 20 ms, a large portion of which is spent context switching between the kernel and TAL/RaTP, and between TAL/RaTP and DSMC. Moving TAL/RaTP into the Unix kernel would eliminate the additional context switching, and we are currently investigating such an implementation. A semaphore V operation costs only 16.5 ms since it is non-blocking, i.e., the issuing process continues without waiting for the final acknowledgment from the remote DSMC.

Get or discard (8K bytes)			
without forwarding:	43.4 ms	throughput:	185 Kbytes/s
with forwarding:	63.7 ms	throughput:	126 Kbytes/s
V operation:	16.5 ms		
P operation:	31.5 ms		
Activate segment:	23.3 ms		

Table 2: Measurements of DSMC operations on Unix

As mentioned in §6, we experimented with using socket IPC primitives and System V semaphore primitives to synchronize the TAL/RaTP process and its clients. The numbers reported in this section are from the implementation that used the socket IPC primitives. When using System V semaphores, the average cost of an 8K bytes `get` request is almost 20 ms more than the cost reported in Table 2. We believe the difference is due to the System V implementation of the semaphore primitives, since the two implementations differ only in the code that synchronizes the TAL/RaTP process and its clients.

Page-in (8K)	35.2 ms
Segment activation/deactivation	25.1 ms

Table 3: Ra to Unix Communication Using DSM

7.2 Ra

The DSM partition on Ra for handling local requests is complete. This partition communicates with the DSM implementation on Unix to obtain the remote segments. At this point DSMC is not fully ported to Ra to entertain remote segment requests. Table 3 summarizes the preliminary measurements of a Ra node communicating with a Unix node using DSM. Once again the dominant cost in both segment activation/deactivation and page-in is the context switch time at the Unix end. Currently, a null round-trip message time from one Ra node to another through the ethernet is 3.2 ms. These measurements are from an unoptimized implementation, and therefore there is scope for bringing down the message cost. We will have measurements of Ra nodes communicating with one another using DSM shortly.

8 Conclusions and Future Work

We presented an architecture of a distributed shared memory system and described an implementation of the system in the context of the Ra kernel. We also described and reported on the performance of an implementation of the system on Unix. Detailed algorithms for the DSMC primitives, and simulation studies comparing these primitives to RPC are presented in Reference [20]. The utility of these primitives in programming distributed algorithms is illustrated in Reference [23]. So far, our work has concentrated on the mechanisms of distributed shared memory. As part of our future work, we intend to gain more experience with the system and address policy issues such as when to use the RPC mechanism and when to use the DSMC primitives, where to place the semaphore segments, and how to recover from failures.

References

[1] Mustaque Ahamad, Mostafa H. Ammar, José M. Bernabéu Aubán, and M. Yousef A. Khalidi. Locating resources in a distributed system using multicast communication. In *Proceedings of the 13th Conference on Local Computer Networks*, October 1988. (Also available as Technical Report GIT-ICS-87/44).

[2] François Armand, Frédéric Herrmann, Michel Gien, and Marc Rozier. Chorus, a new technology for building unix systems. In *European UNIX systems User Group Autumn Conference*, pages 1–18. EUUG, October 1988.

[3] José M. Bernabéu Aubán. *Locating Resources in Distributed Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, December 1988.

[4] José M. Bernabéu Aubán, Phillip W. Hutto, and M. Yousef A. Khalidi. The architecture of the Ra kernel. Technical Report GIT-ICS-87/35, School of Information and Computer Science, Georgia Institute of Technology, December 1987.

[5] José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. Clouds — a distributed, object-based operating system: Architecture and kernel implementation. In *European UNIX systems User Group Autumn Conference*, pages 25–38. EUUG, October 1988.

[6] José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. The architecture of *ra*: A kernel for *clouds*. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.

[7] K. P. Birman, T. A. Joseph, F. Schmuck, and P. Stephenson. Programming with shared bulletin boards in asynchronous distributed systems. Technical Report 86–772, Cornell University, Dept. Of Computer Science, August 1986.

[8] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[9] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[10] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190–197, May 1986.

[11] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM '86*, August 1986.

[12] P. J. Denning. On modeling program behavior. In *Proceedings of the Spring Joint Computer Conference, Volume 40*, pages 937–944, Arlington, Va., 1972. AFIPS Press.

[13] D. Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[14] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feburary 1988.

[15] R. Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[16] M. Yousef A. Khalidi. *Hardware Support for Distributed Object-based Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, June 1989. Technical Report GIT-ICS-89/19.

[17] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. J. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5), November 1983.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[19] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. In *Ninth Conference on Principles of Programming Languages*, 1982.

[20] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-ICS-88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988.

[21] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *18th International Conference on Parallel Processing*, St. Charles, Ill, August 1989. To appear.

[22] Umakishore Ramachandran and M. Yousef A. Khalidi. A measurement-based study of hardware support for object invocation. *SOFTWARE — Practice & Experience*, 1989. To appear. Also available as Technical Report GIT-ICS-88/21.

[23] Umakishore Ramachandran and M. Yousef Amin Khalidi. Programming with distributed shared memory. In *Proceedings of COMPSAC '89*, September 1989. To appear.

[24] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[25] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSII)*, pages 31–39, Palo Alto, CA, October 1987.

[26] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[27] Sun Microsystems. *Unix Interface Reference Manual.* Mountain View, California, 1986.

[28] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating Systems Review*, 22(4):25–34, October 1988.

[29] Christopher J. Wilkenloh. RaTP: A transaction support protocol for *ra.* Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989. In Preparation.

[30] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.

[31] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 13–24, November 1987.

An Object-Oriented Implementation of Distributed Virtual Memory*

Gary M. Johnston
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, Illinois 61801–2987

E-mail: johnston@cs.uiuc.edu, roy@cs.uiuc.edu

Abstract

Distributed virtual memory allows processes executing on different computers interconnected by a network to share virtual memory. This permits parallel programs written using shared memory to execute on a loosely coupled multiprocessor (or distributed system) without modification. It emulates a true shared-memory multiprocessor, using the paging hardware and operating system virtual memory page-fault handling support to trap accesses to non-resident pages. The pages may be on local secondary storage (as in a conventional paging system), or they may be stored remotely (on another processing node in the network). A non-resident page that is remote must be fetched using network services. Unlike traditional paging systems, networked computers supporting processes which share virtual memory must cooperate to maintain the coherency of the shared pages.

To date, distributed virtual memory has not been extensively evaluated. Several policies and mechanisms require further experimental investigation including those relating to page and process migration. In order to study such problems further, distributed virtual memory has been implemented as part of *Choices*. One of our goals is to evaluate distributed virtual memory as a mechanism to support memory-mapped access to distributed system services. This mechanism could be used to replace the more "traditional" schemes that use messaging or RPC [2].

The *Choices* project [3, 4, 16] is a study of the use of object-oriented and class hierarchical design techniques to design and implement complete operating systems and families of operating systems. *Choices* is written in C++, and all operating system policies and mechanisms are represented within the framework of a *class hierarchy*.

In this paper, we describe what we believe to be the first object-oriented implementation of distributed virtual memory. The implementation uses a page-oriented, state-based, event-driven network protocol. This protocol handles lost packets using an underlying low-level unreliable datagram networking service. It allows efficient, lightweight communication between nodes for the purpose of negotiating access to shared distributed virtual memory pages.

*This work was supported in part by NASA under grant number NSG1471 and by AT&T METRONET.

1 Introduction

Distributed virtual memory supports shared memory on networked multiprocessor computers. Processes executing on different computers in such a distributed system access shared memory transparently, relying on the underlying operating system virtual memory paging support to make the data available when (and where) needed. Access to a non-resident page is trapped by the operating system which communicates with other nodes to obtain the required data. Each system must also coordinate access to shared pages in order to maintain the coherency of their virtual memories.

The technology permits the porting of shared-memory parallel applications to a distributed multiprocessor environment without major modifications. If the application involves loosely-coupled communicating processes, the performance of the application on the system may approach its performance on a shared-memory machine. This, by itself, is sufficient motivation to explore practical implementations of the approach. This paper describes the object-oriented design and implementation of distributed virtual memory in *Choices* [6, 7, 9, 14, 16, 17, 18], an object-oriented operating system.

Distributed virtual memory may also provide an efficient and powerful mechanism to support the provision of distributed system and user provided services. In *Choices*, these services are provided by entities called interaddress-space *server objects*. An application process requests access to a service from a *nameserver*. The nameserver grants the request by returning a *proxy object*. The application invokes methods on the proxy object as if it were the server object. Method calls to the proxy object are trapped by and validated in the kernel. The virtual address space of the process is changed to include the server object, and the appropriate server object method is invoked. In a distributed virtual memory environment, the server object may be remote. In this case, either the server object is mapped into the process' virtual memory, or the process can be migrated to the remote node. The data corresponding to the virtual memory mappings are then transferred using the distributed virtual memory support. So, another goal of our implementation is to evaluate whether distributed virtual memory can support distributed system and user services efficiently.

1.1 Issues

Distributed virtual memory is a relatively new concept [11]. Below, we note some of the issues arising from its use.

Page Coherence Strategies. Concurrent updates to a shared page are a potential source of data inconsistencies. One solution is to impose a "single-writer or multiple-readers" constraint on accesses to a page. At any given time, exactly one of the following conditions are permitted for each shared page:

1. The page resides in the memory (or secondary storage) of a single node, and the node has write access to the page; or

2. copies of the page reside on one or more nodes, and each node has read-only access to its copy of the page.

Page coherence strategies involve a negotiation between nodes for honoring the above constraints on page access. Questions that must be resolved concerning the negotiation include:

- How is a non-resident page (or a copy of the page) located?

- How does a node obtain exclusive write access to a page?

Conceptually, a solution is to have each page "owned" by one node. The owner of a page authorizes write access or the copying of read-only copies of the page matching the constraints above[1]. When a remote computer requires access to a page, it negotiates with the owner of the page. Various page ownership strategies have been studied and it appears that a "dynamic distributed ownership" algorithm is efficient [13]. In this algorithm, page ownership is transferred to the requester when write access is requested. In any algorithm where ownership may change dynamically, the method for locating the page owner may impact performance.

Page and Process Migration Policies. Process and page migration policies attempt to maintain processor utilization, concurrency, throughput, and response time while reducing network traffic. Migration decisions may be based on network page traffic and load distribution information (or estimates, depending on the mechanism). Page and process migration policies are intimately related in distributed virtual memory systems and separation of the two concerns may be counterproductive. Also, these policies may both depend upon the application. For example, a distributed parallel application might perform best with a page migration policy, and an application requiring access to geographically distributed system services via distributed virtual memory might perform best with a process migration policy. The Cross-Architecture Procedure Call [8] demonstrates the value of process migration in certain cases. See §1.2.

Network Protocol. A specific low-level network protocol is needed to provide the specialized support required by an efficient distributed virtual memory implementation. This is a major part of our research effort. We propose a low-level, page-oriented, state-based, event-driven protocol for node-to-node page coherence messaging. This protocol may have benefits in high-speed networks in which the software processing of high-level protocols becomes a communications bottleneck. This is discussed in §4.

1.2 Previous Work

IVY (Li, Yale). Little work has been done in the area of distributed virtual memory. IVY was the first known implementation of such a system [11, 12, 13]. IVY was implemented on a network of up to 8 Apollo workstations running modified Aegis operating systems. The coherence strategy was implemented at the user level, with assistance from the modified kernel. Benchmark results were promising, showing that in many cases shared-memory parallel applications can achieve significant performance improvement when executed in a distributed virtual memory environment, as opposed to a single uniprocessor. Li's prototype used a dynamic distributed page ownership strategy, utilizing a simple RPC protocol between nodes for coherence message and coherency negotiation.

Mach (Rashid, *et al.*, CMU). Mach [1] allows multiple processes ("threads") to execute within a UNIX address space [20]. In addition, Mach supports a two-way interface between the kernel and "external backing store managers" (or "pagers") [21]. Usually, these pagers represent secondary storage or backing store for the pages of a virtual address space (or a

[1] Although in some algorithms (See [13, page 236], for example), the page owner may give other nodes the authority to distribute read-only copies of a page.

part of a virtual address space). The interface provides kernel-to-pager routines that are invoked by the Mach kernel to request pages of data from the pager and to write back modified pages. In addition, pager-to-kernel routines are invoked by a pager to indicate to the kernel that requested data has been provided or to request that a page mapping be invalidated. This kernel↔pager interface allows specialized external pagers to participate in supporting distributed virtual memory between Mach nodes. While this design has been described in the literature [21], no information on an actual implementation has been reported.

The Cross-Architecture Procedure Call (Essick, University of Illinois). Conceptually, the Cross-Architecture Procedure Call (CAPC) [8] is a combination of distributed virtual memory and the Remote Procedure Call (RPC) [2]. CAPC is useful in situations where there exist two loosely-coupled processors, one of which is considerably faster than the other, that share a common data representation. For example, the processors might belong to a single-user workstation and a high-performance compute-server interconnected by a local-area network. An application executing on the workstation arranges with the operating system to map the *code* for certain compute-bound procedures onto the compute-server. When the workstation application attempts to execute a procedure mapped onto the server, a page fault occurs. The actual execution of the procedure takes place on the compute-server. Pages of data are transferred from the client to the server on demand, similar to a distributed virtual memory system. When the procedure is completed, control returns to the workstation and any results are paged back on demand. Unlike distributed virtual memory, page consistency in CAPC is not a problem because the client and server are never executing code of the application at the same time.

CAPC can provide a very efficient remote procedure call implementation if access to large data structures can be localized within the procedures executed on the server, if the data involves dynamic structures like linked lists, or if the volume of data paged to the server is small compared with the number of server accesses to that data.

Copy-On-Reference Process Migration (Zayas, CMU). Zayas [22] investigated the use of "copy-on-reference" to reduce process migration costs. When a process is to be migrated to a new node, only process control information is transferred. Initially, the pages of the virtual address space are *not* moved. Rather, the pages are transferred on demand as they are referenced by the process executing on the new node. A distributed virtual memory system would subsume this approach. Zayas showed that after a process migrates, it accesses a relatively small number of pages that were resident previously in its virtual address space. Therefore, this method saves much of the data transfer overhead that would be incurred if the entire address space were moved at migration time. Zayas did *not* investigate process migration *policies*, and his work did not involve the migration of processes which share memory.

1.3 The Choices Project

This section provides a brief overview of the *Choices* project [6, 7, 9, 14, 16, 17, 18]. *Choices* began as an investigation of the use of class hierarchies and object-oriented design for the construction of multiprocessor operating systems [3, 4]. We believe that we have been mostly successful [14, 16, 18]. All operating system concepts and components are implemented within the framework of a class hierarchy, subclasses of which encapsulate machine

dependencies and separate mechanisms from policy decisions. *Choices* is implemented in over 30,000 lines of C++ and supports paged virtual memory management, interrupt and exception handling, parallel processing, heterogeneous file systems, and networking. Our experience is that C++ supports class hierarchies and object-oriented design, while not sacrificing efficiency [19].

We now exploit these object-oriented design techniques to facilitate the construction of customized operating systems for shared-memory and loosely-coupled multiprocessors and distributed systems. *Choices* is being used in the Tapestry[2] laboratory [7] to support the investigation of parallel applications and distributed systems in a heterogeneous environment. It currently runs on the Encore Multimax shared-memory multiprocessor, and is being ported to the Intel iPSC/2 hypercube.

2 Virtual Memory in Choices

This section reviews virtual memory management in *Choices* [5, 18] in preparation for the subsequent sections that discuss distributed virtual memory.

2.1 The Virtual Memory Class Hierarchy

In *Choices*, operating system concepts are specified as abstract classes that are organized within a class hierarchy. Implementations of these concepts are represented as instantiations of concrete classes that specialize the behavior of the abstract classes. The class hierarchy includes, for example, virtual memory management, process management, scheduling, exception handling, and file systems. Specializations of the classes for process management include implementations for the National Semiconductor NS32332 and Intel 80386 processors. Inheritance facilitates much code reuse.

Figure 1 shows the most important classes in the *Choices* hierarchy that are concerned with virtual memory management. These are discussed below.

2.1.1 The Choices Process Model

In *Choices*, a thread of execution is represented by an instance of class Process. A Process specifies the state or context of the thread of execution including its virtual address space. Processes are scheduled and executed by invoking the methods **add** and **remove** on ProcessContainers. Subclasses of class ProcessContainer represent schedulers, wait queues, and processors. Figure 1 shows a FIFOScheduler and two implementations of the class CPU.

The **add** method on a FIFOScheduler is used to store a Process in an instance of the container. A FIFOScheduler can store many instances of a Process. The **remove** method extracts a Process from the FIFOScheduler according to the FIFO scheduling discipline. Several other schedulers have been implemented, but are not shown in Figure 1. These include a preemptive round-robin timeslicing scheduler and several different instances of the "universal scheduling system" [10].

Class CPU is an abstract class representing concepts common to its subclasses. The **add** method on a CPU dispatches a Process, while the **remove** method preempts the contained Process. Two hardware-specific concrete subclasses are shown in Figure 1. Each physical processor in a multiprocessor is represented by an instance of a subclass of CPU.

[2]Tapestry is funded by an NSF CISE grant.

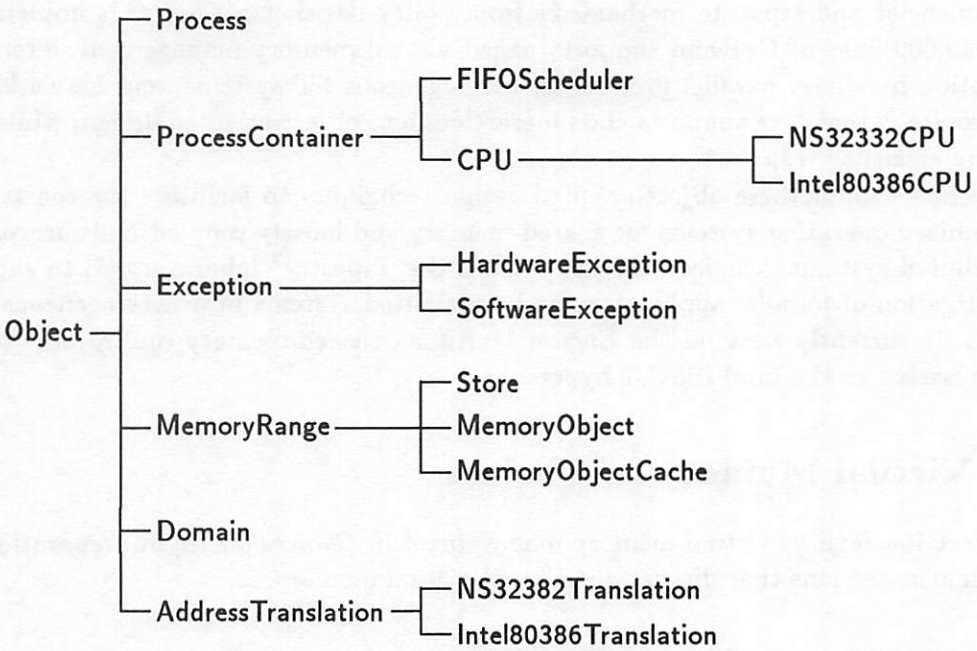


Figure 1: The *Choices* Virtual Memory Class Hierarchy

Processes are usually moved between ProcessContainers by the **handlingFunction** of Exceptions. A *Choices* Process is preempted (**removed** from its CPU) by the **handlingFunction** of an Exception that has been **raised** on that CPU. Subclasses of SoftwareException provide methods by which an executing Process can suspend its own execution. Examples include terminating itself and blocking on a semaphore's **P** method. HardwareExceptions represent hardware events including interrupts, time-slice timer expirations, and page faults.

2.1.2 The Choices Virtual Memory Model

Memory management is rooted by the abstract class MemoryRange shown in Figure 1. This class represents contiguous storage of an arbitrary number of units or blocks. A Store represents the physical memory of a machine. If a machine has significantly different types of physical memory, a Store may exist for each type. For example, on a shared-memory multiprocessor with global memory in which each processor also has some additional "private" memory, one instance of Store would represent the *global* memory and other instances of Store would also exist to represent each processor's *private* memory.

Choices supports the concept of a memory object whose behavior is encapsulated by class MemoryObject and its subclasses. The units of a MemoryObject are accessed using the **read** and **write** operations. Subclasses of MemoryObject exist to represent open files and paged virtual memory backing storage.

The virtual address space of a Process is represented by an instance of class Domain. Multiple Processes may share a Domain in shared-memory applications. Conceptually, a Domain maps MemoryObjects into virtual memory so that they may be directly addressed. This conceptual view is shown in Figure 2.

Before a Domain can allow a Process to directly access a page of a MemoryObject, that data

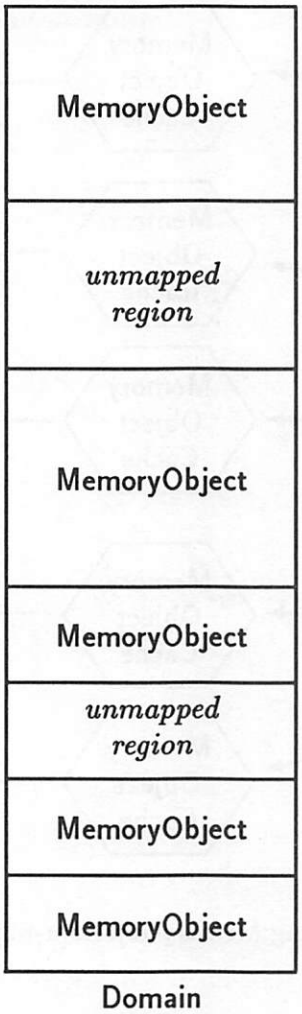


Figure 2: Conceptual View of a Domain

must be resident in physical memory. A MemoryObjectCache is used to manage the caching of pages of a MemoryObject in physical memory. See Figure 3. The MemoryObjectCache maintains a machine-independent mapping of MemoryObject pages and physical memory page frames. That is, a MemoryObjectCache manages the caching of pages of a MemoryObject in physical memory. A Domain maintains a mapping between virtual addresses and MemoryObjects. In this way, a particular MemoryObject can be mapped (by a MemoryObjectCache) in more than one Domain at potentially different virtual addresses.

Figure 4 shows the two kinds of memory sharing possible in *Choices* (besides, of course, no sharing at all). Two or more Processes may execute within the same Domain. This permits lightweight context switching between the Processes. Alternatively, two or more Processes may execute within separate Domains which share one or more MemoryObjectCaches (and the cached pages of the underlying MemoryObjects).

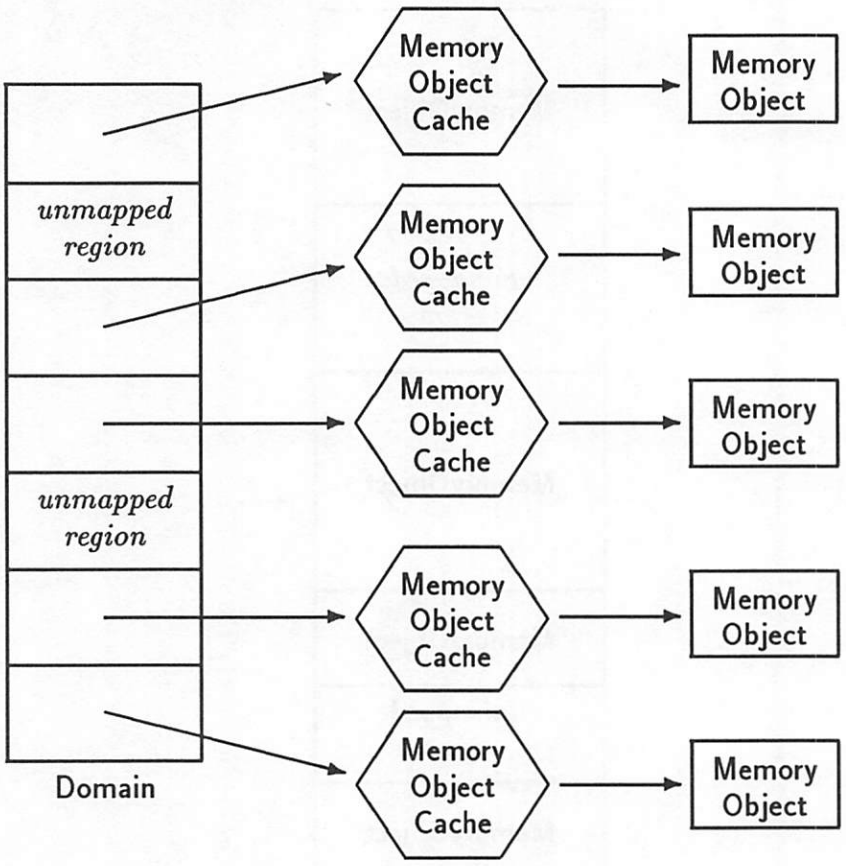


Figure 3: Domain mapping MemoryObjects using MemoryObjectCaches

2.2 Page Fault Handling

Subclasses of the abstract class AddressTranslation represent the hardware-specific tables used to maintain virtual memory address translation. All the machine dependencies of specific hardware are encapsulated within these subclasses. Each Domain is associated with an AddressTranslation, on which methods are invoked to **add** and **remove** virtual memory mappings as necessary.

When a page fault occurs, the Domain determines the MemoryObjectCache corresponding to the faulting virtual memory address. The **cache** method is invoked on the MemoryObjectCache with an argument specifying the page offset (within the MemoryObjectCache) at which the fault occurred. Like the approach pioneered in Mach [15], each MemoryObjectCache maintains a *hardware-independent* mapping of the pages of the underlying MemoryObject and physical memory page frames. The MemoryObjectCache takes whatever actions are necessary to repair the fault. For example, it may allocate a page of physical memory and copy the corresponding data from the MemoryObject into that page. It returns the new physical mapping information to the Domain. The Domain then adds the mapping to its AddressTranslation, and the faulting instruction is retried or restarted.

Because MemoryObjectCaches maintain their own physical memory mapping information, the Domain's AddressTranslation is free to discard machine-dependent virtual address mappings at any time. Such mappings can be easily recovered from the MemoryObjectCache if

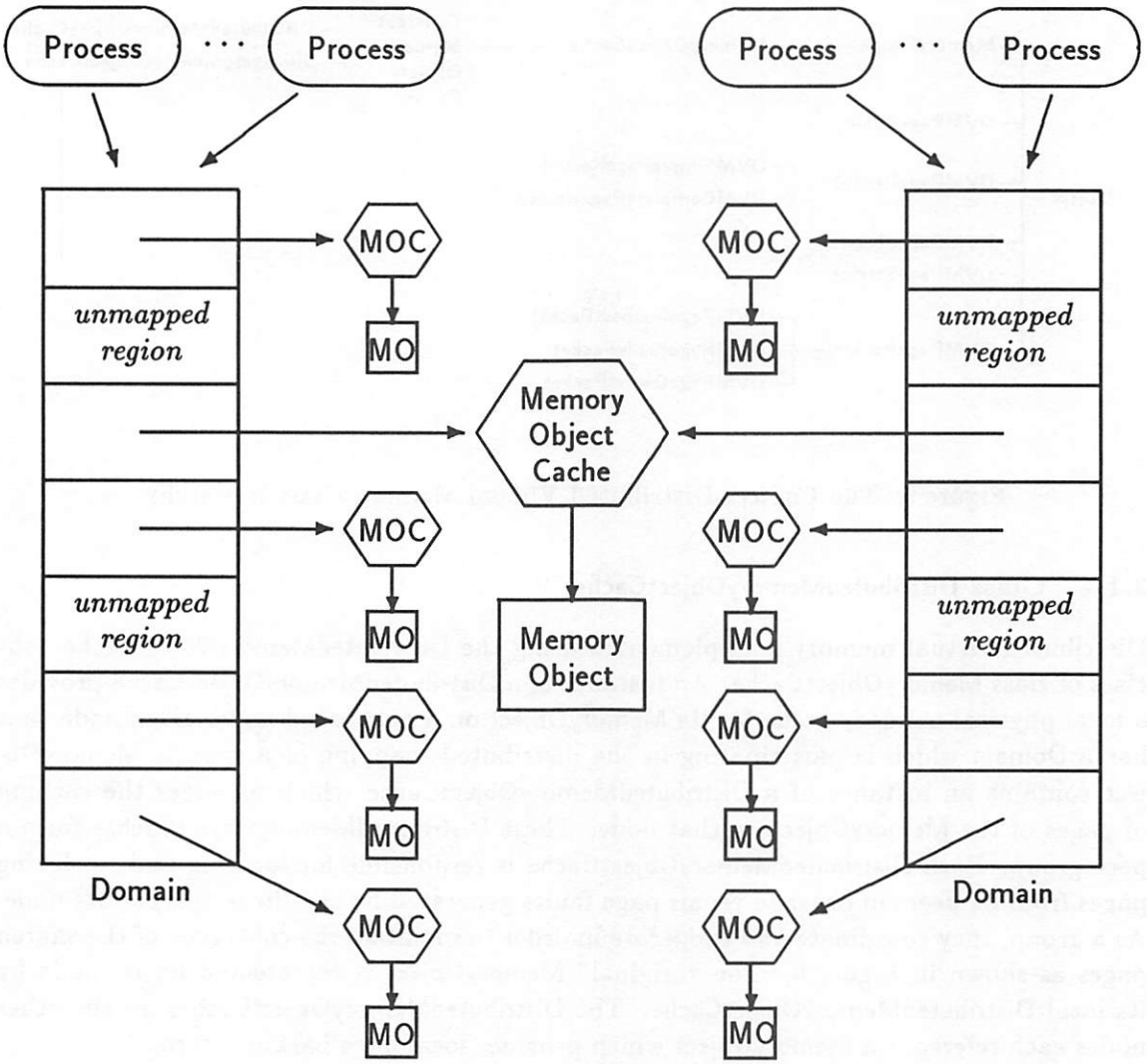


Figure 4: Sharing Memory in *Choices*

and when a fault occurs. It is important to note that the Domain has no need to know or care *how* a MemoryObjectCache repairs a fault.

3 Distributed Virtual Memory in Choices

This section discusses the extensions to the *Choices* virtual memory class hierarchy which implement distributed virtual memory. There are two basic parts of the implementation. First, extensions to the *Choices* virtual memory class hierarchy provide page fault handling and maintain memory coherence. Second, a network protocol specifies how participating nodes communicate in order to maintain memory coherence (see §4).

3.1 The Distributed Virtual Memory Class Hierarchy

Figure 5 shows the distributed virtual memory extensions to the *Choices* class hierarchy.

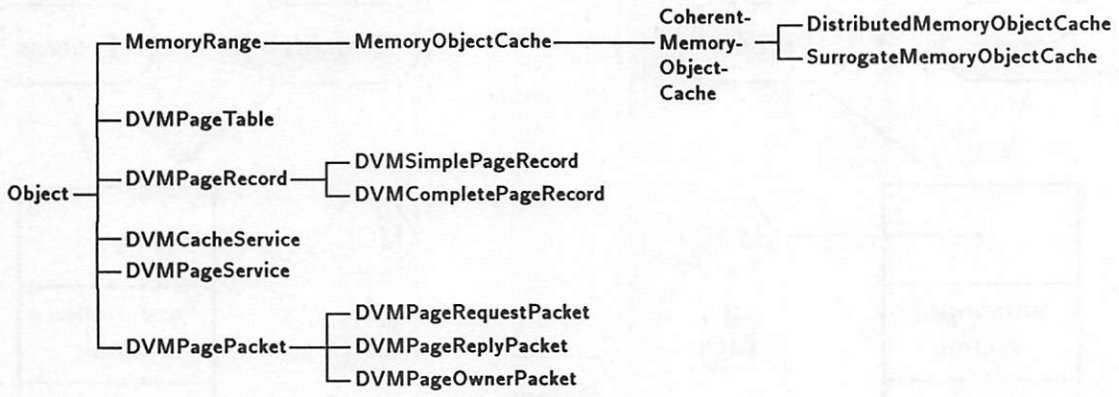


Figure 5: The *Choices* Distributed Virtual Memory Class Hierarchy.

3.1.1 Class DistributedMemoryObjectCache

Distributed virtual memory is implemented using the DistributedMemoryObjectCache subclass of class MemoryObjectCache. An instance of a DistributedMemoryObjectCache provides a local physical memory cache for its MemoryObject on a networked node. Each node that has a Domain which is participating in the distributed mapping of a specific MemoryObject contains an instance of a DistributedMemoryObjectCache which manages the caching of pages of the MemoryObject on that node. These DistributedMemoryObjectCaches form a peer group. Each DistributedMemoryObjectCache is responsible for locating and retrieving pages from its peers in order to repair page faults generated by the Process(es) on its node. As a group, they coordinate and cooperate in order to maintain the coherence of the shared pages as shown in Figure 6. The "original" MemoryObject is represented on its node by its local DistributedMemoryObjectCache. The DistributedMemoryObjectCaches on the other nodes each reference a MemoryObject which provides local page backing storage.

This design has the advantage that none of the other components of the virtual memory system need to be modified in order to share virtual memory across networked nodes. The Domain↔DistributedMemoryObjectCache interface is identical to the Domain↔MemoryObjectCache interface. Therefore, adding distributed virtual memory to *Choices* requires no changes to the *existing* virtual memory class hierarchy.

3.1.2 Classes DVMPageTable and DVMPageRecord

Internally, the most important component of a DistributedMemoryObjectCache is its DVMPageTable. The DVMPageTable contains DVMPageRecords, one per page. A DVMPageRecord records the state of the page it represents. This includes traditional virtual memory paging information such as the page's physical address (if it is resident). It also includes other information dependent on the state of the page. This includes:

- the current "best-guess" as to the owner of a page (if it is not owned locally), and
- a list of remote DistributedMemoryObjectCaches that have read-only copies of the page (if the current node is the owner of the page).

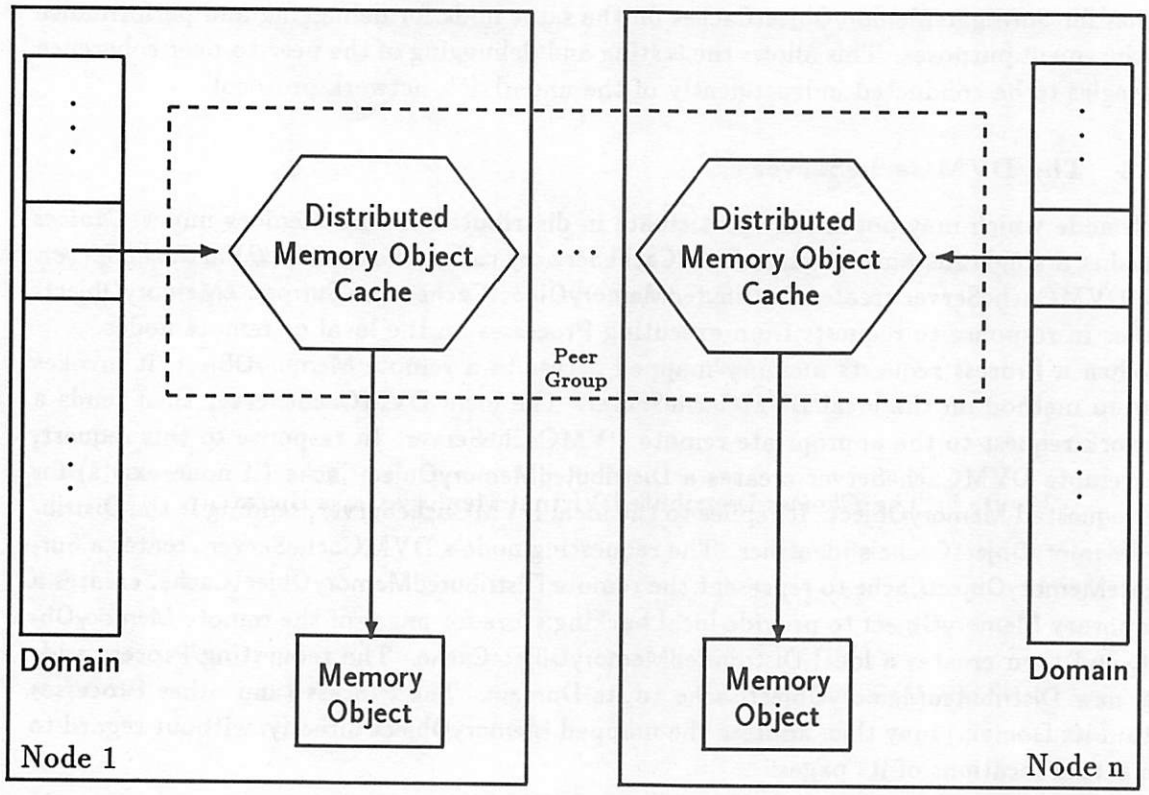


Figure 6: A peer group of DistributedMemoryObjectCaches.

Some of this information is necessary only when the page is in certain states, and is not always present in the DVMPageRecord. For example, if the page is owned locally, the remote owner information is *not* needed, but a (possibly empty) list of remote copy holders *is*. Conversely, if the page is *not* owned locally, the remote copy holder list is *not* needed, but the probable identity of the remote owner *is*.

In addition to providing state information about a page, the DVMPageRecord implements the page coherence strategy. Subclasses of DVMPageRecord implement different coherence strategies. Examples are DVMSimplePageRecord and DVMCompletePageRecord. Since the page state information and coherence strategy are completely encapsulated within the DVMPageRecord, different strategies can be easily substituted. This is discussed further in §4.

3.1.3 Class SurrogateMemoryObjectCache

An instance of class SurrogateMemoryObjectCache represents each remote peer of a local DistributedMemoryObjectCache. The DistributedMemoryObjectCache invokes methods on the SurrogateMemoryObjectCache as if it were a local instance of the remote DistributedMemoryObjectCache. The SurrogateMemoryObjectCache packages the request and transmits it to the corresponding remote DistributedMemoryObjectCache. The SurrogateMemoryObjectCache contains a node address and an identifier. The node address identifies the remote node on which the DistributedMemoryObjectCache exists. The identifier uniquely identifies the DistributedMemoryObjectCache on the remote node.

A pragmatic advantage of this design is the ability to substitute DistributedMemoryObject-

Caches for SurrogateMemoryObjectCaches on the same node for debugging and performance measurement purposes. This allows the testing and debugging of the peer-to-peer coherence strategies to be conducted independently of the underlying network protocol.

3.1.4 The DVMCacheServer

Each node which may potentially participate in distributed virtual memory under *Choices* contains a single instance of class DVMCacheService, called the node's *DVMCacheServer*. The DVMCacheServer creates DistributedMemoryObjectCaches and SurrogateMemoryObjectCaches in response to requests from executing Processes on the local or remote nodes.

When a Process requests memory-mapped access to a remote MemoryObject, it invokes a setup method on the local DVMCacheServer. The local DVMCacheServer then sends a network request to the appropriate remote DVMCacheServer. In response to this request, the remote DVMCacheServer creates a DistributedMemoryObjectCache (if none exists) for the requested MemoryObject. It replies to the local DVMCacheServer, sending it the DistributedMemoryObjectCache's identifier. The requesting node's DVMCacheServer creates a SurrogateMemoryObjectCache to represent the remote DistributedMemoryObjectCache, creates a temporary MemoryObject to provide local backing store for pages of the remote MemoryObject, and then creates a local DistributedMemoryObjectCache. The requesting Process adds this new DistributedMemoryObjectCache to its Domain. The Process (and other Processes within its Domain) may then address the mapped MemoryObject directly, without regard to the actual locations of its pages.

3.1.5 The DVMPageServer

Each node has an instance of class DVMPageService, called its *DVMPageServer*. The DVMPageServer is the low-level interface between the distributed virtual memory objects and the network. It is used by SurrogateMemoryObjectCaches to send packets to their remote DistributedMemoryObjectCaches. It also receives packets destined for local DistributedMemoryObjectCaches. In this case, it looks up the local DistributedMemoryObjectCache using the local identifier specified in the received packet.

4 The Distributed Virtual Memory Network Protocol

This section describes the network protocol used to implement distributed virtual memory under *Choices*. This protocol was designed to use an unreliable datagram service for node-to-node transmission[3]. It is assumed that corrupted packets are identified by the receiver and discarded. The reason for this approach is that we were unwilling to pay the potential performance penalties of a heavyweight reliable protocol. This is discussed later.

4.1 Protocol Model

4.1.1 Message Types

There are several kinds of messages used in the coordination and page coherency maintenance of distributed virtual memory between nodes in a *Choices* distributed system. These are described here.

[3]By "unreliable" we mean that delivery is not *guaranteed*, even though many local area networks are quite reliable in practice.

GetWrite: Sent to request the writable copy (and ownership) of a page from the page's (supposed) owner.

HereWrite: Sent to transfer the writable copy (and ownership) of a page to a node which has requested it.

AckHereWrite: Sent to acknowledge a **HereWrite.**

GetRead: Sent to request a read-only copy of a page from the page's (supposed) owner.

HereRead: Sent to transfer a read-only copy of a page to a node which has requested it.

GetUpgrade: Sent to request write access (and ownership) of a page from the page's owner when the requesting node already has a read-only copy.

HereUpgrade: Sent to transfer ownership of a page to a node which has requested it.

AckHereUpgrade: Sent to acknowledge a **HereUpgrade.**

Invalidate: Sent to request that a node with a read-only copy of a page disallow all access to it.

AckInvalidate: Sent to acknowledge an **Invalidate.**

OwnerHint: Sent as a reply to a **GetWrite** or **GetRead** message when the recipient is not the owner of the page. The replying node provides "better" page ownership information to the requesting node.

4.1.2 Packet Types

A message is encapsulated by an instance of DVMPagePacket (or one of its subclasses) as shown in Figure 5. A DVMPagePacket consists of the following fields:

Destination: Specifies the node address and DistributedMemoryObjectCache identifier of the destination.

Source: Specifies the node address and DistributedMemoryObjectCache identifier of the sender.

Type: One of the message types listed above.

Unit: Specifies the page within the DistributedMemoryObjectCache.

Each instance of a DVMPagePacket (or one of its subclasses) contains these fields. Message types **AckHereWrite, GetUpgrade, HereUpgrade, AckHereUpgrade, Invalidate,** and **AckInvalidate**require no further fields.

In our environment, the hardware-imposed page size (4096 bytes) is too large to be sent in one network packet (Ethernet). That is, multiple network packets must be transmitted to transfer a page of data. Therefore, message types **GetWrite** and **GetRead** use a DVMPageRequestPacket which contains an additional field that specifies the set of requested page fragments. Message types **HereWrite** and **HereRead** use a DVMPageReplyPacket that includes two additional fields: the page fragment data and the fragment number. This means that each DVMPageReplyPacket actually contains only a fragment of a page.

The message type **OwnerHint** uses a DVMPageOwnerPacket format that includes fields which provide page ownership information.

4.1.3 Page-State Model

From the point of view of each node, at any given time each page is considered to be in one of several states. The state is maintained in the page's DVMPageRecord. Page state transitions are potentially caused by three types of events:

1. a page fault caused by an executing Process on the local node,
2. a message from another node, or
3. the expiration of a timeout on a request sent to another node.

These events are translated into method calls on the page's DVMPageRecord which responds appropriately and handles state transitions. The actual coherence strategy depends only upon which subclass of DVMPageRecord is used. Different subclasses may be easily incorporated merely by changing which subclass of DVMPageRecord is used by the DistributedMemoryObjectCache (via its DVMPageTable).

4.1.4 Advantages of this Approach

A state-based, event-driven protocol has advantages over other types of protocols that might be used. For example, IVY [13] used a simple RPC protocol. Consider what happens when the owner of a page receives a request for write access to a page. If the owner has distributed read-only copies of the page to other nodes, it must invalidate each of them before giving ownership of the page to the requester. Using RPC, each of these invalidations must be requested (and responded to) serially. That is, after the RPC request is sent to the first node in the list of copy holders, its response is waited for before sending the request to the second node in the list, and so on. Using our state-based, event-driven protocol, however, all **Invalidate** requests may be transmitted immediately, and **AckInvalidate** responses can be accepted in any order.

Another advantage is that the protocol handles lost messages directly, without relying on a reliable underlying network protocol layer. This means that errors due to lost messages can be handled in a more efficient, lightweight, protocol-specific way.

Finally, consider page disassembly/reassembly. This is necessary in our environment because the page size (4096 bytes) is larger than the largest allowed network packet ($\approx$1500 bytes using Ethernet). If our messaging were based on UDP (or even IP directly), and if we sent a whole page of data in a single DVMPageReplyPacket, allowing the IP layer to disassemble and reassemble the packet, then if a single fragment of the IP packet were lost, the IP layer would have no choice but to discard all of the received fragments. We would then have to request the page again, and all of the fragments would be retransmitted. By handling page reassembly directly as part of the distributed virtual memory networking protocol, if one or more fragments are lost, we can request that only those page fragments not yet received be retransmitted (using the DVMPageRequestPacket).

4.2 A "Read/Write Access or No Access" Example

Table 1 is the state transition table for a simple example coherence strategy. It allows nodes to have either read/write access or no access to a given page. It does not provide for the distribution of read-only pages. The effect is as if all page faults were treated as write faults. The entry for a state/event shows the response message sent ("-" means "no message

Current State	Event					
	Page Fault	GetWrite	HereWrite	AckHereWrite	OwnerHint	Timeout
NoCopy	GetWrite I0	OwnerHint -	AckHereWrite -	- -	- -	- -
I0	- -	- -	AckHereWrite OwnedReadWrite	- -	GetWrite -	GetWrite -
OwnedReadWrite	- -	HereWrite I1	AckHereWrite -	- -	- -	- -
I1	- -	HereWrite -	AckHereWrite -	- NoCopy	- -	HereWrite -

Table 1: A "Read/Write Access or No Access" Example

sent") and the new page state ("-" means "no change"). A timeout is armed whenever an intermediate state (I0 or I1) is entered (or reentered). In Table 1, the transmission of a **GetWrite** means that the node requests only those page fragments which it has not yet received. The transmission of a **HereWrite** means that the node transmits all of the requested fragments.

4.3 An Example That Allows Read-Only Pages

Table 2 is the state transition table for a more complete example that allows read-only copies of pages to exist when requested. The entries mean the same as above. A timeout is armed when entering (or reentering) an intermediate state (I0–I7). The transmission of an **Invalidate** message means that an **Invalidate** message is sent to each node in the list of copy holders. When an **AckInvalidate** is received, the sending node is removed from the copy holder list. In Table 2, the receipt of an **AckInvalidate** means that the last outstanding **Invalidate** message has been acknowledged. The transmission of a **GetWrite** or **GetRead** means that the node is requesting only those page fragments that it has not yet received. The transmission of a **HereWrite** or **HereRead** message means that the node is transmitting all of the requested fragments.

5 Conclusion

This paper has described an object-oriented implementation of distributed virtual memory in *Choices*, an object-oriented operating system. The current state of the implementation is that the "read/write access or no access" protocol given in the previous section has been used to test and debug the extensions to the *Choices* operating system on two networked Encore Multimaxes. The more complex protocol that allows the distribution of read-only pages is nearing completion.

Instrumentation is being installed in the various elements of the distributed virtual memory classes to allow detailed performance analysis. Preliminary measurements indicate that the use of a DistributedMemoryObjectCache to map a MemoryObject on a single node imposes little or no performance degradation as seen by the application. Table 3 shows some preliminary performance measurements of the distributed virtual memory implementation running under *Choices* on two Encore Multimaxes on a moderately loaded Ethernet. It shows that the access time of a page resident on a remote node is a little over 2.5 times more expensive than an access to a page on the local disk, and that the access time of a page that is paged out on a remote node (that is, the page resides on the remote node's

Current State	Event				
	Read Fault	Write Fault	GetWrite	HereWrite	AckHereWrite
NoCopy	GetRead I2	GetWrite I0	OwnerHint -	AckHereWrite -	- -
ReadOnlyCopy	- -	GetUpgrade I6	OwnerHint -	AckHereWrite -	- -
OwnedReadWrite	- -	- -	HereWrite I1	AckHereWrite -	- -
OwnedReadOnly	- -	Invalidate I7	Invalidate I3	AckHereWrite -	- -
I0	- -	- -	- -	AckHereWrite OwnedReadWrite	- -
I1	- -	- -	- -	AckHereWrite -	- NoCopy
I2	- -	- -	- -	AckHereWrite -	- -
I3	- -	- -	- -	AckHereWrite -	- -
I4	- -	- -	- -	AckHereWrite -	- -
I5	- -	- -	- -	AckHereWrite -	- -
I6	- -	- -	- -	AckHereWrite -	- -
I7	- -	- -	- -	AckHereWrite -	- -

Current State	Event				
	GetRead	HereRead	GetUpgrade	HereUpgrade	AckHereUpgrade
NoCopy	OwnerHint -	- -	- -	AckHereUpgrade -	- -
ReadOnlyCopy	OwnerHint -	- -	- -	AckHereUpgrade -	- -
OwnedReadWrite	HereRead OwnedReadOnly	- -	- -	AckHereUpgrade -	- -
OwnedReadOnly	HereRead -	- -	Invalidate I4	AckHereUpgrade -	- -
I0	- -	- -	- -	AckHereUpgrade -	- -
I1	- -	- -	- -	AckHereUpgrade -	- -
I2	- -	- ReadOnlyCopy	- -	AckHereUpgrade -	- -
I3	- -	- -	- -	AckHereUpgrade -	- -
I4	- -	- -	- -	AckHereUpgrade -	- -
I5	- -	- -	- -	AckHereUpgrade -	- NoCopy
I6	- -	- -	- -	AckHereUpgrade OwnedReadWrite	- -
I7	- -	- -	- -	AckHereUpgrade -	- -

Current State	Event			
	Invalidate	AckInvalidate	OwnerHint	Timeout
NoCopy	AckInvalidate -	- -	- -	- -
ReadOnlyCopy	AckInvalidate NoCopy	- -	- -	- -
OwnedReadWrite	- -	- -	- -	- -
OwnedReadOnly	- -	- -	- -	- -
I0	AckInvalidate NoCopy	- -	GetWrite -	GetWrite -
I1	- -	- -	- -	HereWrite -
I2	AckInvalidate NoCopy	- -	GetRead -	GetRead -
I3	- -	HereWrite I1	- -	Invalidate -
I4	- -	HereUpgrade I5	- -	Invalidate -
I5	- -	- -	- -	HereUpgrade -
I6	AckInvalidate NoCopy	- -	- -	GetUpgrade -
I7	- -	- OwnedReadWrite	- -	Invalidate -

Table 2: An Example that Allows Read-Only Pages

Page Location	Access Time
Paged out on local disk.	25ms
Resident on remote node.	65ms
Paged out on remote disk.	120ms

Table 3: Preliminary performance results

disk) is a little less than 5 times more expensive than an access to a page on the local disk. These results are preliminary in that no attempt at optimization has been made regarding the underlying Ethernet driver and network packet filter interface. Several optimizations are possible, and we expect improved results after they are performed.

Distributed virtual memory appears to have several properties that could be exploited to advantage in new, high-speed networks. It can use a simple protocol that requires less processing. It transfers predetermined fixed-size messages. Finally, given appropriate data packet sizes, it can be used to eliminate the need to copy data from message buffers to user memory since this function can be performed by the memory mapping hardware.

In the future, we plan to complete the implementation of distributed server objects and provide a variety of object-oriented, distributed system services using distributed virtual memory.

References

[1] Mike Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. Technical Report CSL–83–7, Xerox Corporation, October 1983.

[3] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[4] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[5] Roy Campbell, Vincent Russo, and Gary Johnston. A class hierarchical, object-oriented approach to virtual memory management in multiprocessor operating systems. Technical Report UIUCDCS–R–88–1459, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[6] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report UIUCDCS–R–89–1510, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989. Also submitted to the ACM Symposium on Operating System Principles.

[7] Roy H. Campbell and Daniel A. Reed. TAPESTRY: Unifying shared and distributed memory parallel systems. Technical Report UIUCDCS–R–88–1449, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1988. Also Tapestry Technical Report No. TTR88–1.

[8] Raymond Brooke Essick, IV. *The Cross-Architecture Procedure Call.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1987. Also Technical Report No. UIUCDCS–R–87–1340.

[9] Gary M. Johnston and Roy H. Campbell. A multiprocessor operating system simulator. In *Proceedings of the USENIX* C++ *Conference*, pages 169–182, October 1988. Also Technical Report No. UIUCDCS–R–88–1460, Department of Computer Science, University of Illinois at Urbana-Champaign.

[10] Douglas E. Leyens. A Choices implementation of the universal scheduling system. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[11] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986. Also Technical Report No. YALEU/DCS/RR–492.

[12] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, pages 94–101, 1988.

[13] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[14] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A class hierarchy for building stream-oriented file systems. In *Proceedings of the European Conference on Object-Oriented Programming*, 1989. Also Technical Report No. UIUCDCS–R–89–1507, Department of Computer Science, University of Illinois at Urbana-Champaign.

[15] Richard Rashid et al. Machine–independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[16] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1988. Also Technical Report No. UIUCDCS–R–88–1415, Department of Computer Science, University of Illinois at Urbana-Champaign.

[17] Vincent F. Russo. *The Design of an Object-Oriented Operating System.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (forthcoming).

[18] Vincent F. Russo and Roy H. Campbell. Virtual memory and backing store management using object-oriented design techniques. In *Proceedings of the Conference on*

Object-Oriented Programming Systems, Languages and Applications, 1989 (forthcoming). Also Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

[19] Vincent F. Russo and Simon Kaplan. A C++ interpreter for scheme. In *Proceedings of the USENIX* C++ *Conference*, pages 95–108, October 1988. Also Technical Report No. UIUCDCS–R–88–1461, Department of Computer Science, University of Illinois at Urbana-Champaign.

[20] Avadis Tevanian et al. A UNIX interface for shared memory and memory mapped files under Mach. In *Proceedings of the USENIX Conference*, pages 53–67, June 1987.

[21] Michael Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 63–76, 1987.

[22] Edward R. Zayas. *The Use of Copy-On-Reference in a Process Migration System.* PhD thesis, Computer Science Department, Carnegie-Mellon University, April 1987. Also Technical Report No. CMU–CS–87–121.

Experience with Process Migration in Sprite*

Fred Douglis
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
douglis@sprite.Berkeley.EDU

Abstract

This paper reports on experience with the Sprite process migration facility. Sprite provides transparent remote execution to support load sharing through the use of idle workstations. Process migration is used to reclaim workstations when their owners return. On Sun 3/75 workstations, the cost of selecting an idle host and invoking a remote process is about 400 milliseconds. This time is substantially greater than the cost of creating the same process locally, but it is much less than the typical execution time of programs that are run remotely, such as compilations and text formatting. The cost of migrating an active process is a function of the number of dirty pages it has, the number of file blocks that must be flushed from the host's file cache, and the number of open files it has. This time ranges from 110 milliseconds to migrate a small process with no open files, to several seconds to migrate a process with many dirty pages and file blocks and several open files. Remote execution has been used regularly for approximately 9 months to perform compilations in parallel. I draw conclusions about the usefulness of remote execution for parallel compilation, and I present lessons we learned about process migration and system building in general.

1 Introduction

By executing independent tasks in parallel on idle workstations, applications may substantially reduce turnaround time. However, the usefulness of remote execution is limited if processes must be terminated to reclaim a workstation when its owner returns, or if processes behave differently when they are run remotely. Sprite [8] provides a transparent process migration facility to allow noninvasive access to idle workstations. An application invokes a program remotely by performing a system call that combines migration with *exec*, replacing the process's execution image with a new program on the other host. If the owner of the remote host returns, a daemon migrates the remote process back to its own host. The primary client of migration in Sprite is a parallel version of *make* (called *pmake*), which uses idle hosts to perform compilations and other tasks in parallel. This paper discusses the experience we have had with process migration, from experimenting with an initial prototype in 1986-87 to using migration daily over the past 9 months.

*This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and in part by the National Science Foundation under grant ECS-8351961.

The next section provides some background on Sprite's process migration facility, summarizing what has appeared elsewhere [2,3]. In Section 3, I discuss the history of process migration in Sprite, from its initial implementation to its current state. We found that migration was much harder to get working than we had expected, and even harder to keep working as the rest of the system evolved. Once migration was in daily use, however, changes in the system that affected migration were noticed immediately and corrected.

Section 4 analyzes the performance of remote execution and process migration using four metrics: the time to invoke a remote program, the time to migrate a process after it has been executing at length, the execution penalty due to transparent remote execution, and the overall speedup of application programs using remote execution to perform tasks in parallel.

Section 5 considers the lessons we have learned from implementing and using process migration over a period of time. From an implementation standpoint, we found that file system bookkeeping was the hardest aspect of process migration to get right, and we found that transparency could be provided with low overhead as long as important operations are location-independent (particularly interactions with the file system). I also draw lessons about systems in general: for example, a feature such as process migration must be used periodically if it is to work despite changes to the system.

In Section 6, I conclude the paper and discuss current and future work.

2 Goals and Design

This section summarizes the goals and design of Sprite's process migration facility. I define some terminology used throughout the paper. I then discuss the means by which transparency is supported during remote execution, and the mechanism for migrating active processes.

The primary goals of process migration in Sprite are transparency and noninvasiveness. Sprite provides *transparency* by making processes appear in all ways to execute on a single host throughout their lifetimes. The host on which the process appears to execute is termed its *home*, and the host on which it physically executes at any given time is its *physical host*. If the process's physical host is different from its home, then it is executing *remotely*. Sprite provides *noninvasiveness* by migrating a remote process during execution if its host becomes unavailable, leaving no residual dependencies on the remote host after migration. Finally, I refer to the host initiating process migration as the *source*, and the recipient of the process as the *target*.

In order to support transparent remote execution, Sprite has several relevant characteristics:

- **Shared file system.** The system has a single file system namespace, so a file name refers to the same object regardless of location. (Section 3 below discusses the poor performance of remote execution when the same name can refer to different objects on different hosts.) Processes can access files and devices on remote hosts transparently.

- **Inter-process communication through the file system.** Communication with other processes is performed using file system objects such as pipes and *pseudo-devices* [13]. Pseudo-devices are used for system services such as the X Window System and access to the internet, for which location transparency would otherwise

present a problem. By using the file system for communication with the internet server, processes appear to internet hosts to be on a single host throughout their lifetime; the Sprite file system automatically forwards communication between the internet server and a remote process as necessary.

- **Location-transparent system calls.** All system calls by a remote process that depend on its location are forwarded to its home host for evaluation. Calls that interact with remote processes, such as sending signals, are redirected from the home to the physical host as needed.

- **Transparency to the user.** A remote process appears in a listing of processes on its home and retains the same process identifier throughout its lifetime. The parent-child relationships between processes are maintained regardless of where they execute, with all synchronization of exiting processes performed on the home host. Furthermore, the home host alone is responsible for knowing the current location of all processes that are tied to it; this host is similar to the LOCUS "origin site", which is the host on which a process is created [9]. However, the home host in Sprite is inherited, so children of remote processes behave as though they were created on the same host as their parent.

Processes are migrated by encapsulating their state on the source and transferring the state to the target via kernel-to-kernel remote procedure calls (RPC). The transfer cost is typically dominated by the time to send the process's open files and virtual memory to the target. To encapsulate the state of an open file, the kernel sends information about the file itself (its unique identifier, including which server stores the file, and state depending on the type of the file encapsulated) and the process's stream for the file (*e.g.*, the offset into the file, and the mode in which the file is accessed). File transfer is costly primarily because of Sprite's file system cache consistency algorithm (described in detail in [7]). Read-only files are cachable on multiple hosts simultaneously, and if a file is read and written by only one host then that host may cache the file. However, any time a file is open for writing on one host while another host accesses the file for reading or writing, the file is cached only by the server storing the file. If a file is cached by a host and then caching for the file is disabled, dirty blocks for the file must be flushed to the host storing the file, and clean blocks are discarded. When a process migrates, any files it has open for writing are briefly open for writing simultaneously on multiple hosts, and caching of those files is disabled. Measurements of the cost of cache flushing are presented below in Section 4.

To transfer a process's virtual memory, Sprite writes the process's dirty pages to a shared file server. The pages are retrieved from the server as the process page-faults. By comparison, Locus, V [11], and Charlotte [1] transfer the entire address space, which may take orders of magnitude more time than transferring the rest of the process's state. Accent addresses the "process migration bottleneck" by transferring virtual memory in a lazy fashion: the target of the migration retrieves memory from the source as it is referenced, thus amortizing the cost of memory transfer over the execution of the process [14]. Although lazy virtual memory transfer makes the act of migration faster than direct memory-to-memory transfer, it requires that the source of a migration dedicate memory to the process after the migration has completed. When a Sprite workstation is reclaimed, all resources used by foreign processes are relinquished as the processes are migrated back to their home host.

3 History of Implementation Effort

The path to a usable migration facility was long and difficult, but in retrospect was worth the effort. Migration was first implemented in Sprite in 1986, and we were able to perform measurements of its performance in the 1986-87 academic year. Our initial measurements suggested obvious areas for improvement, most notably in the area of distinguishing between location-dependent and location-independent operations. The original implementation forwarded nearly all system calls home, including calls that involved locating files, because each host maintained a distinct prefix table that mapped file system domains to servers [12]. Rather than keeping copies of the prefix table consistent between multiple hosts, naming was performed on the home host using its prefix table. Forcing naming operations to be redirected via the process's home slowed down compilation benchmarks by approximately 20%. In fact, there was no particular reason to permit the same prefix on different hosts to refer to different domains, and we solved this performance problem by legislating the equivalence of prefix tables among multiple hosts.

Although migration worked well enough to perform simple tests at this point, some features were missing: certain types of files, such as pseudo-devices, could not be encapsulated; there was no automatic host selection, so tools such as *pmake* could not yet take advantage of migration; and there was no recovery, so the failure of a host with a foreign process could affect other processes (or the kernel) on the process's home as well. Using migration on a regular basis had to await changes to fix these problems.

While we implemented additional functionality relating to process migration at the user level, the file system underwent major changes to add recovery after hosts reboot. The changes to the internal state associated with each file caused file descriptor encapsulation to become entirely unusable. Because migration was not yet in regular use, we were not even aware that the changes presented a problem until we tried working with migration again in the fall of 1987. The file system was about to be redesigned to fix a number of problems, including issues relating to process migration, so process migration itself was put on hold pending the file system changes. Those changes were completed in late spring of 1988, at which point work on process migration resumed.

Getting migration working again was difficult, mostly due to interactions with the reorganized file system. Bookkeeping between file servers and migrating processes on client workstations proved to be extremely complicated, compared to the rest of the migration facility. In particular, locking and updating the data structures for an open file on multiple hosts simultaneously provided numerous opportunities for deadlocks, race conditions, and inconsistent reference counts.

Once the reintegration with the file system was complete, we were able to implement and test the other missing pieces—error recovery and host selection—and we started using migration regularly in the fall of 1988. Regular use provided the opportunity to find and correct some additional problems that did not arise with simpler test cases. More importantly, the few changes to the rest of the system that impacted process migration were detected almost immediately and corrected.

4 Performance

Many more remote processes execute to completion than are evicted, so the user's view of the system is affected more by the overhead of remote invocation and execution than by the time to migrate an active process. The most important measurements for remote execution are the time to select an idle host, the time to start a program on another host, and the performance penalty incurred by executing remotely rather than locally. The success of remote execution may be evaluated by the overall performance improvement from parallel execution of actual applications on idle hosts. On the other hand, the success of eviction depends upon the degree to which Sprite meets its goal of noninvasiveness: in practice, the time to evict all foreign processes from a workstation is on the order of a few seconds, during which workstation owners do not appear to notice any obvious degradation in performance. Section 4.1 discusses remote execution, and Section 4.2 discusses eviction.

4.1 Remote Execution

To start a program remotely in Sprite, a process obtains the use of an idle host and then performs a remote *exec* to invoke the remote program. Methods of selecting hosts for distributing load, with and without process migration, have been discussed at length in the literature (*e.g.*, [6,11]). Sprite uses a shared file that contains the load average and idle time of each host, as well as information about the number of foreign tasks currently using the host. To find an idle host, a process uses a library routine to lock the shared file, select a host appropriate for offloading (low load average, idle for at least five minutes, and no foreign tasks currently using it), update the count of foreign tasks, and unlock the file. When the host is no longer needed, the file is locked while the entry for the host is updated again. Sprite currently takes approximately 160 milliseconds to select and release a host, running on Sun 3/75 workstations, because all accesses to the file require network remote procedure calls.

State transfer for remote invocation is much like migration, except that no virtual memory is transferred. It currently takes 188 milliseconds on Sun 3/75's to *fork* locally, *exec* a process on a remote host with the standard set of three file descriptors (standard input, standard output, and standard error) and no dirty file blocks, and wait for the remote process to exit; this compares to 86 milliseconds when the *exec* is performed locally. Additional overhead from open files and dirty file blocks is discussed below in Section 4.2.

The total time to select an idle workstation and start a program on it compares favorably to the cost of other remote execution facilities, such as the Digital Systems Research Center distant process (**dp**) facility [10]. **Dp** takes 1 second on Firefly workstations (using multiple MicroVAX-II processors) to start a new distant process. However, **dp** takes 6 seconds to initialize before being usable, so the SRC parallel *make* facility does not use **dp** unless enough tasks may be offloaded to amortize the overhead. The cost in Sprite is relatively constant, and *pmake* will offload tasks any time idle hosts are available, even if only one task is executed at a time. By offloading tasks whenever possible, Sprite minimizes the effect of CPU-intensive operations on interactive response.

The degradation due to remote execution depends on the ratio of location-dependent system calls to other operations, such as computation and file I/O. Figure 1 shows the total execution time to run several programs, listed in Table 1, both entirely locally and entirely on a single remote host. One might expect remote execution to be slower than

Name	Description
pmake-P	recompile *pmake* source sequentially using *pmake*
ditroff	run *grap* | *eqn* | *ditroff* on a 15000-word document
rcp	copy a 1 Mbyte file to another host using TCP
fork	fork and wait for child, 1000 times
gettime	get the time of day 10000 times

Table 1: Workload for comparisons between local and remote execution.

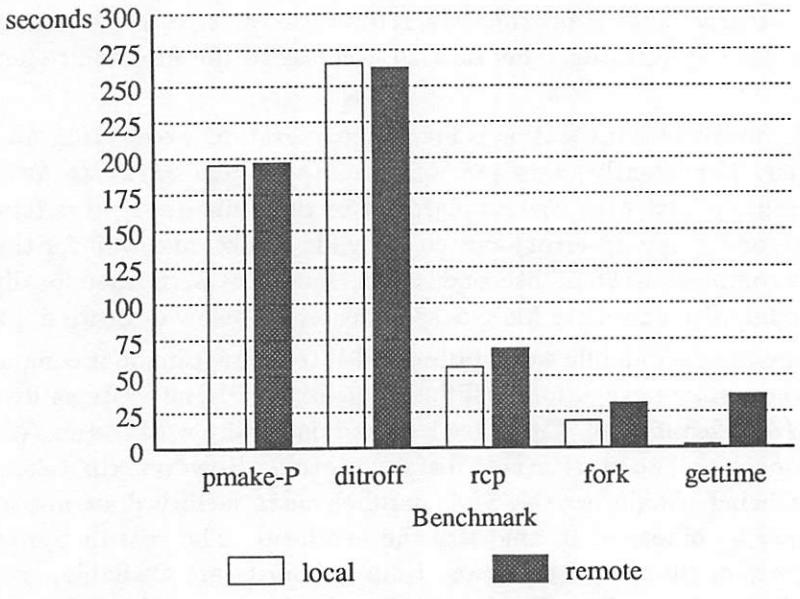


Figure 1: Comparison between local and remote execution of programs.

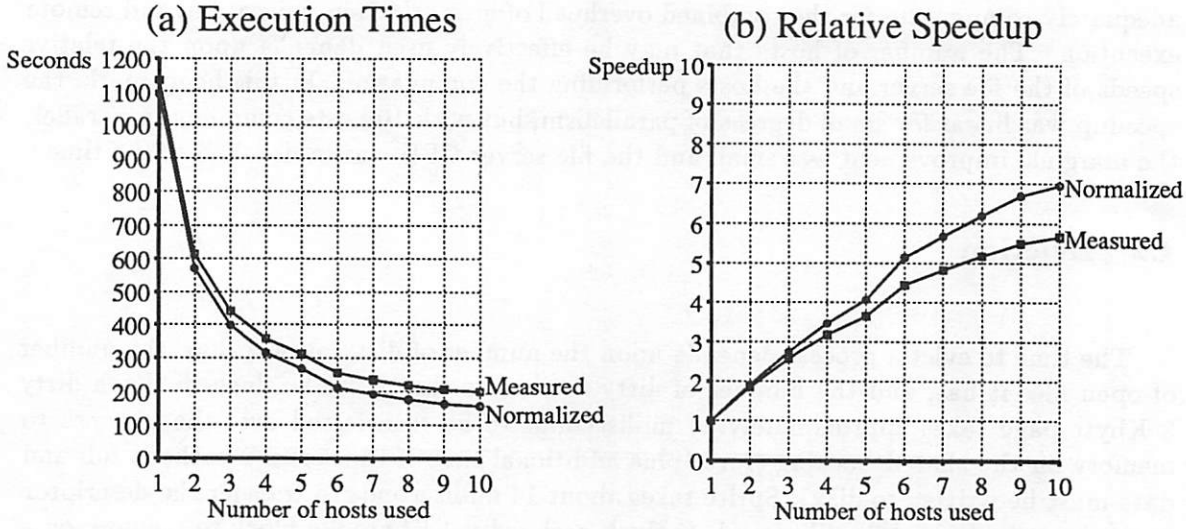


Figure 2: Performance of recompiling the Sprite file system using a varying number of hosts. Each graph shows the measured performance and the normalized (parallelizable) performance. The speedup is the reciprocal of the time saved, by comparison to using a single host.

local execution due to overhead from forwarding location-dependent system calls. As may be seen in Figure 1, however, applications such as compilations and text formatting show little effect from remote execution. In fact, executing the *ditroff* pipeline was slightly faster remotely than locally, due to differences in process scheduling while performing remote procedure calls. The next benchmark, *rcp*, copies data using TCP; it communicates with a user-level TCP server on the home node of the process performing the copy, so forwarding TCP operations to the server on the home node causes *rcp* to perform about 20% more slowly when run remotely than locally. It is also possible for a program to perform many location-dependent system calls without much user-level computation, thereby suffering a large performance penalty from running remotely. The last two benchmarks, *fork* and *gettime*, are contrived examples of this type of degradation.

The usefulness of process migration in our environment may be demonstrated by the performance of the primary application that uses migration, namely *pmake*. Figure 2 shows the total elapsed time to recompile and relink the Sprite file system using a varying number of machines in parallel, and the speedup obtained from using idle hosts. The benchmark consists of 39 independent compilations, followed by loading the resulting object files into a single file. Each migration is performed at the level of a Makefile command (*i.e.*, a single compilation). A new host is requested for each Makefile command and returned to the pool of available hosts when the command is complete. Figure 2(a) includes two curves, showing the measured elapsed times and the same times with fixed overhead removed: starting *pmake* and determining out-of-date dependencies takes about 26 seconds, and loading the object files into a single image takes 17 seconds. Figure 2(b) shows the relative improvement, for both the actual elapsed time and the portion of the compilation that could be executed in parallel. For example, using two hosts was about twice as fast as using one host, while using ten hosts was 5.5 times as fast overall as a single host. Using ten hosts showed a 7-fold improvement for the portion of the compilations that could be parallelized.

Figure 2(b) demonstrates that benefits of using a small number of hosts in parallel

adequately compensate for the combined overhead of host selection, migration, and remote execution. The number of hosts that may be effectively used depends upon the relative speeds of the file server and the hosts performing the compilation. In this benchmark, the speedup was linear for small degrees of parallelism, but with 10 hosts compiling in parallel, the marginal improvement was small and the file server CPU was in use 90% of the time.

4.2 Eviction

The time to evict a process depends upon the number of dirty pages it has, the number of open files it has, and the number of dirty file blocks that must be flushed. Each dirty 8 Kbyte page takes approximately 14 milliseconds to be transferred over the network to memory on the shared backing store (plus additional time if the server's cache is full and data must be written to disk). Sprite takes about 14 milliseconds to transfer the descriptor for each open file, and 7 milliseconds to flush each dirty 4 Kbyte file block to memory on a file server. The total time (in milliseconds) to migrate a long-running process on Sun 3/75 workstations is approximated by the following formula:

$$\text{time to migrate} = 110 + 14s + 7b + 14f$$

s = number of dirty 8 Kbyte pages
b = number of dirty 4 Kbyte file blocks
f = number of open files

For example, to migrate a 1 Mbyte process with 50 dirty pages, 20 dirty file blocks, and 4 open files, Sprite would take 1.0 seconds. If the entire 1 Mbyte address space were dirty, migration would take 2.1 seconds.

5 Lessons

As of this writing, process migration has been in regular use in Sprite for approximately 9 months. We have had the opportunity to reach some conclusions regarding process migration and systems in general:

1. Distributed bookkeeping is difficult.
2. Insulating migration from the rest of the system is difficult.
3. Keeping the right number of idle hosts busy is difficult.
4. Hiding remote execution simplifies applications.
5. Global naming simplifies transparency dramatically.
6. Migration is expensive, to be used only as a last resort.
7. Above all, "use it or lose it."

Distributed bookkeeping is difficult

File system bookkeeping was by far the hardest part of the remote execution facility to implement. Because Sprite file servers maintain state about open files, the server must update its references when a stream to a file changes hosts. The offset with a stream may be accessed by multiple hosts as a result of migration, so the server maintains state for each stream (including the offset) as well as each file. Streams and files have reference counts associated with them, with one reference per host that accesses the stream or file, but different types of files use reference counts in slightly different ways. When a descriptor migrates, the reference count changes depending on what other references to the object exist and on the type of the file. Implementing the code to encapsulate and deencapsulate file descriptors, therefore, required intimate knowledge of the internal implementation of the file system and the state associated with each file.

Insulating migration is difficult

Sprite is not alone in finding that process migration tends to impact the rest of the system and vice-versa. Theimer refers to migration facilities as being "fragile": in an environment in which the kernel is often modified, migration can break unless everyone modifying the kernel keeps the migration facility in step with other kernel changes [11]. Finkel and Artsy, on the other hand, report that they were able to keep migration sufficiently modular to keep changes to migration from breaking other parts of the kernel and changes elsewhere in the kernel from breaking migration [5].

Although file encapsulation proved to be a thorn in the side of process migration for some time, migration has evolved to be generally orthogonal to the rest of the system. Many kernel modules in Sprite maintain state on behalf of each process. Originally, to encapsulate the state of a process, the process migration facility called a predetermined set of encapsulation procedures, one per module, and each module's portion of the process state was transferred in a separate RPC. When a new module was added to the system, migration would break temporarily unless the state of the new module were encapsulated. We therefore changed migration to use a set of "callbacks" into each module to encapsulate its own portion of a process's state. The migration facility on the source requests the size of the encapsulated state of each module, allocates a buffer to hold the collective state, makes the callbacks to encapsulate the state, and transfers the state in a single RPC to the target. New modules may be added to the system by adding an entry to the callback table; changes to existing modules may be performed without affecting the migration facility itself, by updating the module-specific encapsulation routine whenever the format of the process state changes.

Separating the functionality of migration on a per-module basis proved to have a useful side-effect: implementing process migration on a new architecture required only that a small number of machine-dependent state encapsulation routines be rewritten. It took only about half a day to implement migration on the Decstation 3100, given the existing implementation for Sun workstations.

Keeping idle hosts busy

Pmake performs unquestionably well when performing a small number of independent tasks, but large tasks present some problems. On the one hand, the server's CPU is a

bottleneck if too many hosts are used simultaneously. On the other, *pmake* sometimes has trouble using more than a single host. While we can't do much about the server except to get faster and more plentiful CPU's, getting *pmake* to do more in parallel could be beneficial. As an example, the Sprite kernel is stored hierarchically, with each module having its own Makefile and a single Makefile at the top level of the source tree. If *pmake* is invoked at the top level with a high degree of parallelism, permitting it to invoke several *pmake* processes on idle hosts, then those *pmake*s must be careful not to use much parallelism or they will saturate the server. If they are invoked with low parallelism, then a large module will slow down the entire compilation when it is performed sequentially after the other modules are completed. Currently, only one recursive *pmake* is ever performed at a time, so the child *pmake* can use a high degree of parallelism. However, when the child hits a synchronization point, such as loading all the object files in a module into a single image, only one host is used.

Ideally, we would like to be able to build the kernel in parallel with a single *pmake* controlling the degree of parallelism. One module could be compiled in parallel as one or more modules were completing their linking phase. The problem of independent modules is most likely an artifact of the way we chose to structure the source hierarchy before parallel compilation was available, and we have learned our lesson.

Hiding remote execution

If changing a process's location can change the effects of its execution, then users must take special care to use remote execution only when they know *a priori* that a program is location-independent. For example, the V System preemptable remote execution facility is restricted to applications that execute "only operations whose output is independent of the location at which they are executed" [11]. Although compilations and text formatting are location-independent, many other programs are not: for example, what if *rcp* could not run remotely, and a user invoked *rcp* from within a Makefile? In general, any program that one can invoke from *pmake* should be capable of executing remotely and being evicted when necessary. Sprite only restricts processes that map kernel memory into their address space, and processes that are pseudo-device servers, such as the X Window System display manager.

To the users of applications such as *pmake*, remote execution is invisible. The application merely appears to execute much faster than one would expect it to on a single host. If a set of processes is evicted from another host, they immediately start executing on the home host, perhaps with some performance degradation due to sharing the host with other active processes. We hope to implement a mechanism by which processes may be automatically re-migrated to another idle host if they are evicted, but eviction happens so infrequently that the lack of automatic re-migration does not seem to present a problem.

To a user reclaiming his or her workstation, eviction is invisible as well—or it would be if the daemon evicting processes did not announce the eviction in the system log. We found that messages informing the user when eviction takes place promote goodwill, because users can see that their performance is not impacted as a result of foreign processes.

Global naming is a must

When process migration was first designed, each Sprite host was a distinct system with its own file system namespace and its own process identifiers. The simplest method of guaranteeing location transparency was to forward nearly all system calls home, but performance suffered significantly. Over time, Sprite shifted toward making most system calls location-independent: file naming operations go directly to the server for a file system domain, since file names mean the same across multiple hosts; and process identifiers include the process's home host, so a remote process may send a signal using the standard signalling mechanism on its physical host. By reducing the amount of forwarding required to support remote processes, we were able to improve the performance of remote execution while simplifying it substantially.

Migration is expensive

Our experience with the relative costs of remote invocation and migration corroborate the results of Eager, *et al.*, who used a theoretical model and simulation to compare migratory and nonmigratory load sharing. They concluded that migrating processes for load sharing performance does not generally yield significant improvement over policies with only remote invocation, and they suggested that "costlier but simpler" migration may be appropriate if migration is done primarily for purposes other than load sharing (such as permitting workstation owners to reclaim their hosts) [4].

Remote invocation in Sprite is inexpensive enough to provide performance improvements for all but extremely short-lived processes, assuming that the local host is already highly utilized. Migrating active processes, on the other hand, is often measured in seconds rather than milliseconds. The disparity between migrating new processes and processes with many dirty pages and file blocks suggests that migration is unlikely to be useful for dynamic load balancing. As a last resort to guarantee the response time to the owner of a workstation, however, eviction has proved an appropriate use for migration.

Use it or lose it!

Our single greatest mistake when implementing process migration was to let it sit idle while the rest of the system evolved. We did not have the manpower at the time to add the features described in Section 3, but we could have run simple test cases on a regular basis to ensure that problems would be apparent shortly after being introduced to the system. If we had known quickly that the changes to implement file system recovery had affected migration, the recovery support could presumably have been modified in the process of fixing other problems with it. Instead, we were not aware of a problem until well after the changes had become "carved in stone". The changes to support recovery, which involved several data structures that had been designed without taking the possibility of migration into account, would have required too much effort to fix—given that the entire file system was to be rewritten. Instead, when the file system was redesigned, we paid careful attention to the effects of migration and implemented special functionality to handle migration. This functionality could and should have been incorporated into the system at a much earlier point, given that it was ultimately necessary.

Since migration has been in general use, there have been several occasions when changes elsewhere in the kernel caused problems for migration. Because migration is used frequently

for compilations and other tasks, in each case we quickly observed that migration had been affected. By catching the problems quickly, we were able to correct them relatively easily.

6 Conclusions and Future Work

Some time ago, shortly before the file system was to be reimplemented, we had a lengthy discussion about the future of process migration in Sprite. The consensus at the time was that migration was probably a mistake: it was too difficult to implement, and the performance of a single workstation was sufficient for our needs. However, we believed that the marginal cost to put migration into general use was small enough to justify finishing the implementation and giving migration a chance to prove itself.

In retrospect, I may safely say that our initial lack of faith was misplaced. Process migration has evolved from a toy prototype to a mature, extremely useful facility. Users are thankful not only for the significant performance improvement they see when using other hosts, but for the minimal impact other users have on their own workstations.

Our present work with process migration may be divided into three categories: basic support; extensions; and measurement and analysis. Migration is currently usable only on Sun 2, Sun 3, and Decstation 3100 workstations, and only between two machines of the same architecture. We plan to port migration to Sun 4 workstations, and if possible, provide the ability to perform remote *exec*s between machines of different types. The ability to perform heterogeneous remote *exec*s, along the lines of the LOCUS *rexec* system call [9], could considerably expand the pool of idle hosts available to a single program. We would also like to add automatic remigration after eviction to keep eviction from degrading the performance of the home host.

Finally, we intend to instrument the process migration and host selection facilities to evaluate more aspects of the system, such as migration overhead, host availability, and system bottlenecks. Preliminary measurements of the rates of remote execution and eviction suggest that eviction in practice is rare (perhaps one eviction per 50 remote executions) and takes well under a second on Sun 3/75's for typical compilations. Initial measurements of host usage indicate that about one-third of our workstations are available for migration during the day, on average, and over the course of a weekend closer to two-thirds are available. Server CPU utilization is the most likely bottleneck that would affect overall speedup from parallel execution, but we must await faster servers before we can obtain useful measurements of our new client workstations: for example, a Sun 3/180 file server was 50% utilized servicing requests from two Decstation 3100 clients compiling in parallel. Access to the shared file containing host availability may also prove to be a bottleneck, and we are exploring alternative methods for host selection that might scale better with the size and speed of the system.

Acknowledgements

Peter Danzig, John Hartman, Mendel Rosenblum, Mark Sullivan, and Brent Welch provided valuable comments on early drafts of this paper. John Ousterhout has been instrumental in the design of the process migration facility. Michael Nelson and Brent Welch implemented file descriptor encapsulation and provided debugging assistance. Finally, I thank the referees

for their recommendations, which helped to improve the clarity and organization of this paper.

References

[1] Y. Artsy and R. Finkel. Simplicity, efficiency, and functionality in designing a process migration facility. In *The 2nd Israel Conference on Computer Systems*, May 1987.

[2] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[3] F. Douglis and J. Ousterhout. Process migration in Sprite: A status report. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):8–10, Winter 1989.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *ACM SIGMETRICS 1988*, May 1988.

[5] R. Finkel and Y. Artsy. The process migration mechanism of Charlotte. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):11–14, Winter 1989.

[6] P. E. Krueger. *Distributed Scheduling for a Changing Environment.* PhD thesis, University of Wisconsin, Madison, Wisconsin, June 1988. Computer Sciences Technical Report #780.

[7] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[8] J. Ousterhout et al. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[9] G. J. Popek and B. J. Walker, editors. *The* LOCUS *Distributed System Architecture*, pages 73–89. Computer Systems Series. The MIT Press, 1985.

[10] E. Roberts and J. Ellis. parmake and dp: Experience with a distributed, parallel implementation of make. In *Proceedings from the Second Workshop on Large-Grained Parallelism.* Software Engineering Institute, Carnegie-Mellon University, November 1987. Report CMU/SEI-87-SR-5.

[11] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[12] B. B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed filesystem. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 184–189, Boston, Mass., May 1986. IEEE.

[13] B. B. Welch and J. K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *Summer USENIX '88*, pages 37–49, San Francisco, CA, June 1988.

[14] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–22, Austin, TX, November 1987.

Dynamic server squads in Yackos

Debra Hensgen (debra@ms.uky.edu)
Raphael Finkel (raphael@ms.uky.edu)
University of Kentucky
Lexington, KY 40506-0027

Abstract

This paper describes our experience in implementing server squads in the Yackos operating system. A **squad** is a (nearly) homogeneous group of processes that cooperate to provide a service. The group increases and decreases its membership size to meet client demand. **Yackos** is yet another communication kernel operating system designed for low latency, high bandwidth message passing. We have constructed server squads for locks, timeouts, and multicasting. They have been tested on a Yackos implementation running on a Sequent Symmetry multiprocessor. Our experience has helped us design tools to assist in converting code for an ordinary server to that for a squad.

0. Introduction

This paper describes our experience in implementing server squads in the Yackos operating system. A **squad** is a nearly homogeneous group of processes that cooperate to provide a service. The group increases and decreases its membership size to meet client demand. **Yackos**[1] is yet another communication kernel operating system designed for low latency, high bandwidth message passing. We will refer to converting an ordinary server into a squad as **enhancing the server**. We have enhanced servers for locks, timeouts, and multicast service. They have been tested on a Yackos implementation running on a Sequent Symmetry multiprocessor. Our experience has helped us design tools to assist in the enhancement process.

Squads are intended for an environment in which a large number of processors (workstations, multiprocessors, mainframes, or any combination) share a communication kernel that provides high bandwidth communication with very low latency. The principle motivation for squads is responsiveness to clients, not reliability or availability. Our philosophy is that client processes should not be unnecessarily slowed by servers that cannot keep up with demand while unused processing power abounds. The demand for the various servers will vary over time. Therefore, we are not satisfied with any arrangement in which the number of servers in a class (and the way they partition effort) remains fixed. Our design and implementation of squads strives to keep communication and synchronization overhead to a minimum. This goal is necessary to ensure that an added server can improve service to clients.

The first section of this paper describes Yackos, which provides a reasonable environment for exploring squad design issues. The second section outlines some general principles that squads follow. In the third section we describe the design and implementation of two squads. The fourth section contains the design of tools for automating the enhancement process and our experience using those tools. The fifth section discusses our ideas for improving squad efficiency and problems we expect to encounter when porting squads from our pseudo-distributed homogeneous environment to a truly distributed heterogeneous environment. The sixth section surveys related work. We conclude with experimental results.

1. Environment for squads

For convenience, we implemented and tested squads using a version of Yackos[2] that runs on a Sequent Symmetry multiprocessor. We implemented them without using multiprocessor-specific hardware

features.[1] If squads' intended environment had been multiprocessors, we would have used threads as in Mach[3] or virtual address sharing as provided by Barton and Wagner[4].

1.1. Yackos design

Philosophy

Yackos' design was driven by the following two goals.

- To provide a highly usable kernel-process interface with well-defined semantics.
- To achieve blinding communication speed.

Experience has shown that high-level kernel-process interfaces are useless to applications unless they provide exactly the right services[5]. Yackos therefore retreats to the other extreme: an adequate but very low-level interface. The kernel can deliver messages, but it provides no direct support for threads, flow control, or multicast. All these can be layered above the kernel in subroutine libraries or specialized servers for those clients that are willing to pay the additional overhead involved. For this reason, we often refer to the kernel simply as the **message passer**.

Yackos achieves blinding speed by a combination of tricks and principles. First, context switches between processes and the kernel are largely avoided by placing requests and responses in a data area shared by both. This data area is protected by requiring that processes that need to access shared data employ a package of **interface routines** we provide. As a result, most classical kernel requests (such as "send a message") are procedure calls that do not change context. Second, hints [6] are used to make the usual case fast, as discussed later. Third, we strive to avoid any need for locks or semaphores to protect data shared by the kernel and the process. Last, every feature we considered for the kernel was scrutinized and rejected if it made the usual communication scenario slower.

Processes

A process is an address space and an execution context. Each process has a unique identifier, consisting of its home machine and a sequence number. Each process has an **interface area** that it shares with the kernel. Part of it is read-only to the process, while the rest of it may be read and/or written by either the process or the kernel. Instead of making service calls, a process writes information into this area. This information will be used by the message passer at the next opportunity. In a uniprocessor implementation, that opportunity arises during clock interrupts or the generic kernel call, `NoOp`, which means "Do whatever I have requested in the interface area". One other kernel call is available; a process may call `Block` to be blocked from running until the kernel makes some change to its interface area.

Communication semantics

Processes communicate by sending messages. Small (32 bytes) and large (1 page, exact number of bytes varies between 1k and 8k bytes, depending on the machine) messages are available. (The design also calls for unspecified-length messages, but we will ignore them here.)

To prepare for sending and receiving messages, a process initializes buffer pools in the interface area by calling the interface routine `InitPool`. The kernel needs to know about these pools (shown in Figure 1), since it will help in maintaining them and will empty/fill them automatically. Interface routines help manipulate these queues. `[Lg,Sm]GetOutput` removes a free output buffer of the appropriate class from its queue and returns it. `PutOutput` places a buffer onto the busy output queue to be delivered by the message passer at its convenience. `GetInput` is called to receive a message; it returns a buffer taken from the busy input queue. `[Lg,Sm]PutInput` returns a free input buffer to the proper queue.

The message passer places incoming messages in input buffers it takes from the free input queue. After a message has been passed, the sender's buffer is returned to its free output queue. If both processes are on the same machine, page-mapping tricks may be used to achieve message-passing speed.

There is one exception to this rule. In the timeout server we used the hardware microsecond clock. Members' individual clocks could have been synchronized using techniques we describe in Section 5.

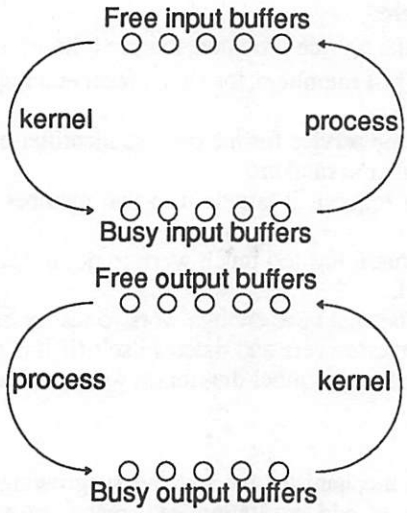


Figure 1: Yackos buffer pools

1.2. Implementation on the Sequent Symmetry

We have implemented the message-passing facilities of Yackos above DYNIX (a 4.2 BSD Unix descendent) on the Sequent Symmetry multicomputer.[2] Both Yackos processes and the message passer itself are implemented as separate Unix processes, as is a reclaimer process that cleans up after terminated Yackos processes. The resulting concurrency enables a Yackos process to continue executing while its messages are passed. Yackos processes are linked with interface procedures for initialization, manipulating queues, and rudimentary name service.

Buffer queues are circular arrays of addresses of message buffers. This data structure (as opposed to a linked-list structure) allows us to avoid locks, since each queue has only one producer and one consumer. For example, only the kernel inserts onto a free output queue, and only the associated process deletes from it.

All processes share a large region of the interface area in which all buffers for all processes are kept.[3] This region allows us to avoid copying large messages. Instead, the message passer places the address of such a message in the busy output queue of the receiver, removes a free input buffer from the receiver, and places that buffer on the sender's free output queue. Small messages are copied.

The message passer continually looks for messages that need to be sent. A circular search of all outgoing message queues results in unacceptable latency between the time a process places a message on such a queue and the time the message passer finds it. This delay worsens as the number of processes grows. To speed the search for messages, we employ a small circular **hint queue**. The interface procedure `PutOutput` writes the process identifier of the sender into the hint queue. Access to this queue is not locked, so entries may overwrite each other and stale entries may remain. Therefore, the message passer checks hints against the appropriate outgoing message queue. It only cycles through outgoing message queues when the hint queue is empty.

Round trip messages (large) can be passed on the Symmetry in just under 200 microseconds. During this period, process **A** sends a large message to **B**, **B** receives it, **B** sends a large response, and **A** receives it.[4] We estimate that if the message passer were not running simultaneously with the processes it serves, round trip message passing would require about 400 microseconds.

---

[2] This implementation also runs on the Sequent Balance multicomputer.

[3] As mentioned earlier, we protect the shared region by requiring processes to use interface procedures to access it. A native implementation could enforce this requirement; under Unix, we cannot.

[4] In this test, neither process puts data in the message or inspects the data that arrive.

2. General squad principles

All squads, regardless of the service provided, tend to follow some common guidelines.

- When members of a squad (just **members** for short) receive a request, they either service it directly or **forward** it to another member.
- When a client queries the name service for the process identifier of a server that is actually a squad, the name service chooses a member at random.
- When a member forwards a request, it targets it to that member most likely to be able to service the request directly.
- When a member perceives that it has too much work to do, it starts an **apprentice** member to which it might assign some of its work.
- If a member decides that it does not have enough work to justify its continued existence, it distributes its responsibilities among the other servers and deletes itself (if it is not the sole remaining member). The members to which the terminating member distributes work are known as the **termination supervisors.**

2.1. Mechanisms

Squads use the following mechanisms for adaptation (growing and shrinking).[5] The **parent**, that is the squad member that desires to add an apprentice, creates an apprentice by means provided by the environment. (In our case, we use Unix `fork` and `exec` calls. In a native Yackos implementation, the parent would negotiate with the kernel to decide where to start the apprentice.) A server usually maintains tables describing current clients and current efforts on their behalf. The parent partitions these tables between itself and its apprentice. It then sends messages to the apprentice describing which member covers which part of the work space (this information may be only a hint), which part of the work space the apprentice is to cover, and details of work in progress now to be covered by the apprentice. In some squads, apprentices can accept new clients before receiving these details. The apprentice sends an update to any clients it has received from its parent. The client (more properly, the subroutine package the client uses to deal with this server squad) uses the update as a hint to target future interactions to the right member. The parent also informs other members of the new apprentice.

We have found it convenient for members to employ two data structures. The **client information table** records details of current interactions with clients. The **forwarding table** describes how work is partitioned among members. When a member receives a client request, the forwarding table produces a good target for forwarding. When a member decides to delete itself, it sends messages to all other members parceling out its responsibilities and indicating how to update their forwarding tables. Partitioning responsibility is server-dependent.

2.2. Growth policy

Each squad needs a policy governing adaptation. Growth is indicated if responsiveness is low and extra processing capability exists. However, growth is not advisable when the entire computing resource is busy. Here we describe a general policy that is used by all squads. Some squads, such as the timeout squad, could use extra information to tune this policy.

Our growth policy is related to Yackos failure semantics. A message will fail if the destination has no available input buffers. Failed reliable messages are flagged and placed on the sender's busy input queue, even if they must be temporarily stolen from its output pool. The frequency of failed messages is valuable information for squad policy. Clients indicate that an attempt failed by setting a flag, called a **busy signal**, in the following message. Squad members count the number of messages with such flags in a (tunable) time interval. If this count exceeds a threshold, the member starts an apprentice.

The busy signal count also serves as a natural back-pressure to discourage growth when the message passer has become a bottleneck.[6] In that situation, the rate of message passing decreases. However, the busy signal count tends to decrease as well, since it is easier for the squad to keep up with the reduced request load. The growth policy will decide not to add apprentices.

---

[5] Some of these mechanisms are specific to our environment, but all can be generalized to other environments.

[6] We assume that even enhancing the message passer cannot improve performance.

Members attempt to delete themselves when the number of times in a row that their input queue is empty exceeds another threshold.

3. Two squads in detail

3.1. The lock squad

First we describe the lock server and then give some details about its enhancement. The lock server uses a lightweight thread package [7] for threads and synchronization. This package can be linked with any Unix process, in particular, with Yackos processes. Each thread has its own stack (carved out of process data area). This package does not provide timeslicing; the currently executing thread continues either until it calls `yield` or it blocks waiting for a resource. Such resources include locks, semaphores, and monitor entrance. When a thread blocks, another thread is chosen in a round-robin fashion. We use monitors and explicit calls to `yield` to let threads yield control when they must wait for lack of buffers. Because of the interplay between this scheduling and buffer management, we created a monitor version of the Yackos interface functions.

Design of the lock server

The lock server is meant to be used by transaction servers, file servers, and other applications requiring locking. It is called upon to acquire, release or alter locks. Locks are identified by unique names; it is the responsibility of the clients to associate these names with resources to be locked. Unique names may be generated by calls to the unique-name server.

To acquire a lock, a client supplies

```
lock name
lock type (read / write)
lock mode (keyed / unkeyed)
lock key (only if keyed)
acquire timeout
hold timeout
```

Any future requests for a keyed lock must contain the correct key. The timeouts are given in seconds and may range from 0 to infinity. The acquire timeout indicates how long the process is willing to wait to acquire the lock. The hold timeout indicates how long the process needs to hold the lock before it may be preempted by another requester.

The lock server acknowledges acquired locks and sends messages to clients whose locks are broken. It also sends messages to clients that cannot obtain a lock within the acquire timeout period. Acknowledgements contain a **cookie** that the client should supply when converting or releasing the lock. A client with the correct cookie (and key, if it is keyed) may change the type, mode, and key of a lock. These conversions are permitted as long as they do not conflict with the way in which clients currently are holding the lock. For conversions, the acquire timeout indicates how long the client will wait to convert the lock before giving up.

The lock server uses a client-information table hashed on lock names. The cookie returned to the client is a pointer to a record in that table. The lock server is a client of the timeout server, discussed below, for both the hold and acquire timeouts. When the timeout expires, the lock server either sends the client a message (''could not get lock within acquire timeout'') or marks the entry as preemptable.

Enhancing the lock server

The lock server uses the general mechanisms and policies for squads described above. However, some details are server-dependent.

Growing

The client information table is treated as a circular list. Each member owns a region of that list, that is, a range of hash values. When a parent creates an apprentice, it divides its region into two equal parts and gives the apprentice one of them. The apprentice can tell from its region that it is not the sole member of the lock squad. It begins by waiting for details from its parent. First the parent tells it how the table is partitioned and which member is in charge of each region. Then the parent sends the contents of the region

of the table being given to the apprentice. After this information is sent, the apprentice becomes a full member. Until then, the apprentice rejects any incoming requests, but it does communicate with the timeout squad and send messages to its newly acquired clients if timeouts demand. Until the apprentice becomes a full member, the parent delays incoming requests that need to be forwarded to it.

Programming is easier and execution more efficient if the lock squad members never cancel a timeout. Instead, they mark the client information table entries so that unwanted timeouts are ignored. Otherwise, we found that timeout elapsed notices would be sent simultaneously with cancellations, so the lock squad members had to deal with unwanted timeouts in any case. We found the technique of ignoring obsolete timeouts so useful that we embedded it as an option in the library linked with timeout server clients.

Shrinking

A lock squad member that chooses to delete itself sheds one half of its region of the table to each of the two members serving adjacent regions. After it has received acknowledgements from both of those members, it deletes itself if its region is empty. (If it is the last surviving member, its region will not be empty after the shedding, so it will not terminate.)

3.2. The timeout squad

Design of the timeout server

The client information table in the timeout server is organized according to tree structured timing wheels[8]. Each node represents two slots, each of which can be empty, a timeout, or a pointer to a child node. All the timeouts within a subtree whose root is at level e have values (expiration times) equivalent mod 2^e. At each level of the tree at most one node is distinguished. The distinguished node is determined by the current time. Only timeouts in the subtree headed by a distinguished node may expire at the current tick. Each tick advances which nodes are distinguished.

The timeout server accepts requests to either set or cancel timers. A request to set a timer includes an identifier; the server returns a cookie that points to a node in the tree. Cancellation requests include the identifier and the cookie.

Every tick (10 milliseconds in our implementation), the server advances the distinguished nodes and checks to see whether any timers have expired. It prunes the tree whenever a timeout expires or is canceled.

Enhancing the timeout server

The mechanism for enhancing the timeout server is very similar to that used for the lock server. It depends on the fact that the tree structure is easy to partition. Members maintain a data structure like a timing wheel tree to record the responsibility of all members. Each member is responsible for a single subtree. (Common ancestors are not, strictly speaking, in the responsibility of any member.) Each subtree has a clock rate exponentially related to the depth of the tree. The clocks are kept consistent among the members by referring to a hardware clock available in our environment.

When a parent starts an apprentice, it splits its subtree into two **buddies**, one of which is retained and the other of which is given to the apprentice. When deleting itself, a member gives its subtree to whatever member holds its buddy. If the buddy is itself split (so no one member is in charge of it), it is unlikely that the prospective disappearing member is the least-active member of the squad, so it refuses to delete itself. Unlike lock squad apprentices, timeout server apprentices may accept new clients any time.

We take the position that a slight delay in announcing the expiration of a timeout is not erroneous. Therefore, a timeout member need not forward every message that strictly speaking should have been served by another member; it can often round the time up slightly to fall within its own purview.

Our experience with implementing the timeout squad suggested another way to guide growth policy. Our implementation uses threads within a single process, two of which are the **main thread** and the **ticker**. The main thread sits in a loop waiting for requests; when one is received it creates a new thread to handle it. The ticker advances the clock and checks the tree for timeout expirations. All of these threads pass control by explicit calls to `yield` or by blocking in monitors. Ticks are delivered through Unix software interrupts, which might occur when any thread is active. The interrupt handler advances a tick count.

The tick count indicates how far behind demand the ticker is getting. If the member is heavily loaded, the tick count steadily increases. Until the timeout server grows (or its clients becomes less active), the ticker cannot catch up with the interrupts. Therefore, in squads similar to the timeout server, policy is influenced by the rate at which some task that should be done periodically falls behind. Conversely, if the threads continue to call `yield` without performing work, the member deletes itself.

4. Enhancement tools: Design and experience

Our experience manually enhancing servers led to a set of routines that can be linked with any server to help it function as a squad. New routines are also provided for the clients to properly interact with the enhanced server. We specify the form that the server and client should take and describe libraries that are linked with the client and server code. We give a description of a multicast squad and relate our experiences using the tools to build that squad.

A major goal in designing the squad tools is to build as much mechanism in as possible, while providing only default policies that can be easily modified or replaced by the server implementor.

4.1. Expected format of client and server

We placed one requirement on the clients of our hand-enhanced servers; they must be able to accept asynchronous messages. This restriction presented no problem because we implemented the clients using a thread package. A production environment supporting squads would likely also contain a language, perhaps similar to Lynx[5], that would make accepting asynchronous messages easy. A programmer using our enhancement tools can choose whether to require the clients to accept asynchronous messages.

The enhancement tools require that the client and server not use the first four bytes of (the data part of) a message, which are reserved. These bytes are reserved for use by squad members to classify inter-squad messages.

Below we present the general form of a server to be enhanced with our tools.

```
InitPool(appropriate parameters);
   /* Interface function;
   allocates Yackos message buffers */
t_init();  /* initialize with thread package */
SquadInit(appropriate parameters);
   /* described below */
ListMe(ServerName);  /* list with Yackos name server */
for ( ;; ) { /* forever */
   if ((MessagePtr = SquadGetInput()) != NULL) {
      switch (MessagePtr->Request) {
         case ReqType1: ...
         case ReqType2: ...
            .
            .
            .
         default: ...
      } /* end switch */
   yield();  /* thread package call */
   } /* if there was a message */
}/* try to get another message now */
```

We expect server implementors to use analysis or a simulation program, to estimate good thresholds for the initial growth and termination policies. The thresholds and, in fact, the policy module itself, can be

[7] For concreteness, we give examples in C, although we don't restrict servers to that language.

changed while a squad is running.[8]

4.2. Server and client stub enhancement tools

Routines for the client

Clients receive messages through the routine `ClientGetInput`, rather than directly through the Yackos routine `GetInput`. This routine filters out behind-the-scenes negotiations between squads and the client.

Four routines, `ResendSquadRequest`, `ChangeAddress`, `TranslateName`, and `LookUp`, help the client find a squad member by maintaining the `AddressTable`.

`TranslateName` is called by a client to get a process identifier (address) for a squad member, given the name of the squad. If a search of the `AddressTable` fails, it calls `Lookup` to assist. `Lookup` uses the Yackos name service to return an address chosen at random from the currently registered members of the squad.

`ChangeAddress` is called by `ClientGetInput` when the reserved bytes contain the flag `CHANGE_ADDRESS`. This message, containing both an old and new address, is received from a squad member that has changed its responsibilities. `ChangeAddress` replaces the old address by the new address in the `AddressTable`. Our squads send such messages as hints to clients to reduce the amount of forwarding.

The Yackos message passer returns messages sent to non-existent processes with the `RECIPIENT_TERMINATED` flag set. `ClientGetInput` passes any such message to the routine `ResendSquadRequest`. `ResendSquadRequest` calls `LookUp` to update the `AddressTable` and resends the message with a new address found by calling `TranslateName`. `ResendSquadRequest` is the absolute to back up the `ChangeAddress` hint; it is needed because clients are not stopped while the squad shrinks. If the original address is not in the `AddressTable`, the client's message was not destined for a squad member, so the message is returned from `ClientGetInput`. (The client must be allowed to decide what to do because a peer has disappeared.)

Enhancement routines for the server

An enhanced server needs to import and export various routines and variables. Some are obligatory for all squads, while the rest are optional.

Exported procedures

The server exports a subset of the following procedures. The description of the procedures below contains only required parameters. The optional parameters are too detailed for this overview.

```
void Split(wholeListPtr, myListPtr, yourListPtr)
RespDesc_t *wholeListPtr, *myListPtr, *yourListPtr;
```

> Build two responsibility descriptors from the original, partitioning the space of responsibility between them. There are no restrictions on the format of a responsibility descriptor.[9] The enhancement routines store the responsibility descriptors in tables and pass them to exported routines but do not interpret them. A responsibility descriptor is used by the squad to record which member is in charge of which sorts of client requests. In the hand-enhanced squads, the responsibility descriptors are descriptors of part of the client information table, but there is no reason why this must always be the case. For example, in a type-setting squad, the descriptors might be integer variables where one value means that a member is responsible for processing mathematical equations, while another is responsible for picture processing. `Split` is automatically invoked when the enhanced server's policy module chooses to form an apprentice.

---

[8] Currently, policy modules must be bound at link time. In an environment providing distributed upcalls[9], the policy module could be supplied at runtime.

[9] One example of the optional parameters that we largely omit in this chapter are pointers to two integer variables. A variable described in the next section is set to indicate whether responsibility descriptors are fixed- or variably-sized. If they are variably-sized, the function `Split` returns the lengths of `myListPtr` and `yourListPtr` in the optional parameters.

```
void Join(oldListPtr, addListPtr, newListPtr)
RespDesc_t *oldListPtr, *addListPtr, *newListPtr;
```

Join two responsibility descriptors into one. `SquadGetInput` automatically invokes `Join` when a member receives a region of the work space from a member that is about to delete itself.

```
Item_t *Delete(listPtr)
RespDesc_t *listPtr;
```

`SquadGetInput` automatically invokes this routine to send information from the client information table to either an apprentice or a termination supervisor. `ListPtr` points to the apprentice's responsibility descriptor or a descriptor for information to be given to the termination supervisor. The member deletes (or copies) one or more elements from its own client information table and returns a pointer to those elements. The member may modify `listPtr` to help keep track of the next value to be returned by `Delete`; it is only a copy of the descriptor to be given to the apprentice. A null pointer is returned if no more elements are to be sent to the member whose responsibility is described by `listPtr`.

```
void Insert(elemPtr)
Item_t *elemPtr;
```

`SquadGetInput` automatically invokes `Insert` when an apprentice receives information from its parent or the termination supervisor is receiving work from a terminating member. It inserts the given elements into the receiver's client information table.

```
int RightVenue(listPtr, messagePtr)
RespDesc_t *listPtr;
LgMessageType_t *messagePtr; /* type defined in Yackos */
```

`SquadGetInput` automatically invokes this Boolean function, possibly repeatedly, when a client request arrives to determine to which member to forward it. It returns true only if the member whose responsibility described by `listPtr` can serve the request contained in `messagePtr`.

```
ReceiveBroadcast(msg)
char msg[];
```

`SquadGetInput` automatically invokes `ReceiveBroadcast` when messages are delivered to a member because another member made a call to `Broadcast`.

```
int GrowthPolicy(messagePtr)
LgMessageType_t *messagePtr; /* type defined in Yackos */
```

This function is called automatically after the Yackos function `GetInput` by `SquadGetInput`. Our default policy counts the busy signals as well as the number of times that `GetInput` returned that there were no messages, clearing these counts at the end of a time interval. These counts help the policy determine when to grow and shrink the squad. `GrowthPolicy` returns a flag with some of the values `CREATE_APPRENTICE`, `ATTEMPT_TERMINATE`, or `USE_MESSAGE` set. If the policy routine sets the `USE_MESSAGE` flag, `SquadGetInput` will process the message as though it is either a squad maintenance message or a client request; otherwise it will take appropriate action if any and free the buffer containing the message.

Exported variables and constants

Servers export some variables and constant definitions to the enhancement library. `RespDescSize` and `KnowledgeEltSize` are records specifying the size in bytes of a responsibility descriptor and the size in bytes of an element returned from `Delete`, respectively. They may be either `VARIABLY_SIZED` or `FIXED_SIZE`; in the latter case, the size must be given.

`RespDesc` is a descriptor whose value indicates the responsibility of the member. It is passed as an argument to an apprentice. The variables `PolicyTimeInt`, `CreateAppThresh`, and `DeleteSelfThresh` are used by the default policy module. A server implementor may also specify a value for `MinThreads`, which is the maximum number of threads that may still be active when a squad

terminates. This integer defaults to one but may need to be set higher, as in the timeout squad.[10]

Imported routines

```
LgMsgType_t *SquadGetInput() /* type defined by Yackos */
```

We require that servers accept messages through the imported `SquadGetInput` routine, as demonstrated in the server structure above. `SquadGetInput` returns null if there is no message to be processed by the switch statement; otherwise it returns a pointer to a message. In an enhanced server, `SquadGetInput` filters out squad-maintenance messages and deal with them behind the scenes. It also silently forwards client requests.

```
void InitSquad(argc, argv, MyName, serviceName)
int argc;
char *argv[];
ProcessIdentifier_t MyName;  /* type defined by Yackos */
char *serviceName;
```

This routine initializes variables private to the library and lists the new member with the name server. The member should call this routine after completing Yackos initialization.

```
void Broadcast(msg)
char *msg[];
```

This routine reliably broadcasts a message containing `msg` to all other members of the squad.

4.3. Operation of the enhanced server

Initialization

A squad member begins by executing the enhanced server code. The thread of the parent that asks the operating system to start the apprentice supplies the parent's process identifier (address) and `RespDesc` as arguments. A non-null parent address indicates that a member is an apprentice; otherwise the process is the only squad member. A member can use `RespDesc` to decide how to initialize its private variables and tables. An apprentice first accepts only messages from its parent of type `KNOWLEDGE_INFO`, acknowledging the last of them. It then accepts only messages from its parent that tell it how to initialize its forwarding table. Finally, the member lists itself with the Yackos name service and enters its main receive-reply loop.

SquadGetInput

`SquadGetInput` screens incoming messages, handling some itself, buffering others, forwarding others, and returning others. After calling `GetInput`, the Yackos routine to receive messages, `SquadGetInput` first calls the `GrowthPolicy` routine. If `GrowthPolicy` suggests `CREATE_APPRENTICE` and this member is not currently starting an apprentice nor attempting to terminate, it starts a thread that creates an apprentice and sends it the necessary information. If the member is attempting to terminate, the termination attempt is aborted. If the `ATTEMPT_TERMINATE` bit is set and the member is not attempting to terminate, a thread is started to contact a termination supervisor and pass messages to it. If the `USE_MESSAGE` bit is set, `SquadGetInput` processes the message further; otherwise, the message buffer is freed.

In this way a server-implemented growth policy routine that accepts advice from outside the squad can be used. Intra-squad messages such as requests to add or delete a member from the forwarding table are handled by `SquadGetInput` itself. Some of these messages cause server-exported routines to be called. For example, a request to accept work from a terminating member becomes a call to `Insert`. `Join` is also used when work arrives from a terminating member. Client requests are tested by `RightVenue` and either forwarded or passed back to the caller.

[10] It can be safe to delete a member of the timeout squad if two threads are still active, the message-filtering thread and the thread that is used to search the tree.

Maintaining forwarding tables

When a member is added or terminates, the parent or termination supervisor, respectively, updates all other members by sending an update message to each known member. The update message is acknowledged. If an apprentice is being added by a parent, **P**, and **P** receives another update from a member **M** before it receives an acknowledgement from **M**, it informs its new apprentice of the update.

Adding apprentices

When a parent hears that its apprentice is started, the thread in charge of creating the apprentice may send information to that apprentice. That thread repeatedly calls the imported routine `Delete` (until `Delete` returns a null pointer) to give the parent a chance to send client information to the apprentice. Then the squad sends a copy of its current forwarding table to the apprentice and updates all other members as described in the subsection above. After receiving acknowledgements from other members for the updates the parent treats the apprentice as it does any other member.

Terminating members

When the policy routine indicates to a member, **T**, that it should terminate, the tools ask some member, **S**, chosen at random to act as a termination supervisor. If **S** refuses (`SquadGetInput` only refuses if the chosen member is itself attempting termination), **T** will not seek another supervisor until the policy routine again tells it to. If **T** still wants to terminate when **S** agrees to serve as the supervisor, **S**'s `SquadGetInput` will call `Join` to determine its new responsibility descriptor. **S**'s `SquadGetInput` will advise all other members to update their forwarding tables, and the tools linked with **T** will call `Delete` to give **T** a chance to share its client information table with **S**. **T** will not accept any new messages at this time. It will destroy its message buffers so that no client requests are lost.

4.4. The multicast squad

A **multicast service** is responsible for delivering messages that are addressed to process groups. Many tools for distributed programming recognize efficient multicast as an important function; the entire ISIS[10] system is built around the multicasting facilities and V[11] provides it in the kernel. Our multicast service allows for various strengths of reliabilities, orderings of messages, and operations on groups. We first describe the multicast service as it would be provided by a single server, then present our design for a multicast squad.

The multicast service

Requests to a multicast server consist of variable length instructions about how a message should be broadcast and the message itself. The first word of a multicast message indicates the request type. Requests are one of `JOIN`, `LEAVE`, `SEND`, or `END_CAUSAL`. If the request is to join or leave a group the only other data in the message is the group identifier and a process identifier.[11] We include the process identifier so that processes can join or leave by proxy. This feature can be useful to parent processes and processes that notice the termination of others. We explain the `END_CAUSAL` below, after the discussion on causal messages.

A request to the multicast service to send a message requires more than just the message to be sent and the group identifier. The sender of such a message specifies **destination type**, **reliability**, and **ordering**. The destination type is chosen from the list: `ONE`, `MAJORITY`, `GROUP`, and `COMBINATION`. The reliability is either `DATAGRAM` or `RELIABLE` and the ordering is one of `UNORDERED`, `ATOMIC` or `CAUSAL`.

If the destination type is not `COMBINATION`, the message is destined for a single group or a subset of a single group. If the type is `ONE`, the sender may specify the destination or allow the multicast server to choose. (The `ONE` to a specific destination can be useful in conjunction with the `CAUSAL` ordering defined below.) If it is `COMBINATION`, the message instructions accompanying the message specify the

[11]The multicast server assumes that a process knowing the name of a group identifier has the right to perform any of the group operations on that group. This server does not care how a process obtains the group identifier. If security issues are a problem, something like the keyed mode in the lock server could be added.

groups involved and some function of them. The function can involve the operators `UNION`, `INTERSECTION`, and `DIFFERENCE`.

All `DATAGRAM` messages are sent without regard to order. The `ATOMIC` and `CAUSAL` orderings provide the same service as ISIS' ABCAST and CBCAST, respectively[12]. `ATOMIC` means that receivers agree on a total ordering of the messages that were sent with that ordering. CBCAST is responsible for most concurrency in ISIS. It is based on Lamport's[13] definition of causal ordering. Every message that could possibly have caused a later message is transmitted to all processes receiving the later message. `CAUSAL` messages have associated `causal_id`'s and the multicast server ensures that a process receiving a `CAUSAL` message has also received all `CAUSAL` messages with the same `causal_id` that came before. A process sending a first `CAUSAL` message tags it as so; processes sending the ensuing messages tag them with the `causal_id`.

ISIS uses garbage collection and a shared set of previous messages at each node to decrease the obviously large messages that would be needed to implement the CBCAST protocol. We expect the last process receiving a `CAUSAL` message with a particular `causal_id` to notify the multicast service via an `END_CAUSAL` request so that old messages can be destroyed.

ISIS also provides GBCAST, which is a broadcast for keeping group members aware of the current membership of the group. That is, in ISIS, all members have a consistent group view. Our multicast server does not bother members of a group with the current membership every time the group changes, but does maintain a group view for the group. When a member joins or leaves a group, the **group view number** is incremented. Each member of the group is associated with two group view numbers, the one when it joined and the one when it left, and a request to join or leave a group elicits a reply with the new group number. When a broadcast is sent to a group, the acknowledgement to the sender contains the current group view number as well as the list of members. The messages are sent only to these members. The acknowledgement is only an agreement that the multicast server will deliver the message as specified; an end-to-end protocol between the sender and receivers can be used to ensure that the messages eventually arrive.

The stubs on the receivers of multicast messages acknowledge reliable messages. They also change the sender field of the message so that the client program thinks that the message came from the message originator.

Multicast server internals

The multicast server keeps information about a group in a table that is hashed on the group name. For each group, the server keeps the process identifiers of the members, their joining and leaving view numbers (if a process joins, leaves and rejoins it is treated as two members),and messages **outstanding** for at least one member of the group. A datagram message is considered outstanding if it has not been sent to all members, a reliable one if it has not been acknowledged. Stored with each group member identifier is an ordered list of identifiers for any outstanding messages to that member. Outstanding messages have the group view number recorded with them as well as which other groups are receiving that message, if any.

Besides information about groups, information about causal broadcasts is also stored. The server stores message pointers in a table indexed by the `causal_id`. The table is large enough so that when the first in a series of causal messages is sent, we expect a free index to be found.

Multicast squad organization

Members divide responsibility only according to multicast groups; they use the broadcast facility in the enhancement tools to handle causal messages. The responsibility is divided as in the lock squad, which also used a hash table for its client information table.

Members join or leave a group by sending a request to any member of the squad. That member forwards the request to the member in charge of that group. When a message to be sent to a group or subset of a group arrives, it is also forwarded to the appropriate member. A member receiving a request to send a message to a combination of groups first checks whether all groups are in its venue. If they are not, it uses the enhancement broadcast routine to send the message to all members of the multicast squad. Each determines its responsibility for the message if any. Responsible members independently deliver unordered broadcasts. Causal broadcasts are delivered as we describe below. Atomic broadcasts require that all responsible members agree on the order of the messages with respect to other messages affecting any of the same groups.

The atomic broadcast protocol is modeled after ISIS' ABCAST. Each responsible member assigns a priority to the message and broadcasts it. The largest priority is chosen and a message is only delivered to a group member if all of the messages with lower priority have been delivered.

Any squad member receiving a request to send a causal message to a group must send all previous causal messages to that group as well. If a member gets a request to send a causal message to a group but does not have the previous causal messages, it uses the enhancement broadcast routine to send a request for the previous messages along with the current message. The possessor of the previous messages broadcasts them; they are cached at all members but are recorded reliably at the requester who then becomes responsible for them. The original possessor can forget about them if it needs the space to record other causal broadcasts.

4.5. Experience suggests an addition to the tools

Our experience with the multicast squad suggests an addition to the tools. What we currently provide is a method for reliably broadcasting messages to all members of the squad. It would have been nice had we provided an alternative call that allowed certain messages to be broadcast to only specific members of the squad. For example, combination multicasts only need to be sent to a subset of the squad members, usually. Similarly, it may be better to simply return the previous causal messages to the member that broadcast the request for them.

5. Extensions

Although experiments on uniprocessors with Yackos indicate that it will be competitive with other communication kernel operating systems, message passing will be slower in a truly distributed environment. It is not clear how well our squad mechanisms will work with very large squads. For these reasons we need to decrease the amount of message traffic caused by adapting (that is, growing or shrinking) a squad. We also need to reevaluate our policy for adapting the squad when faced with the possibility that the client and server will be on different machines.

5.1. Decreasing update traffic

Members of a squad need to stay apprised of the current allocation of work to members. Other information may also need to remain consistent across the squad. For example, members of a truly distributed timeout squad must keep their clocks from drifting too far, and the load-balancing squad needs to maintain a reasonable measure of the total load. We can attack this problem by using an asynchronous dissemination algorithm that mimics broadcast without resorting to a broadcast medium. To use this dissemination algorithm we assume that initially all members agree on a total ordering (0 to $n-1$) of the members. The algorithm itself preserves this condition. First we describe a synchronous dissemination algorithm and explain how it could be used for intra-squad communication if there were global clocks. Then we describe an asynchronous variant that avoids the overhead of synchronizing clocks.

Synchronous dissemination

In synchronous dissemination[14], ticks cycle through the values 0 to $\lceil \log_2 n \rceil$. At tick t, member m sends an update (containing all information that is less than $\lceil \log_2 n \rceil$ old) to member $m+2^t$. If the lack of a message can be interpreted as a null message, null messages need not be sent[15]. Information includes work to be shed, notices about new apprentices, and intentions of members to delete themselves. Within $\lceil \log_2 n \rceil$ ticks (one **cycle**) of new information first being sent out, it is known by all members. Information can be totally ordered by tick initiated and initiator's rank in the squad. Members may use this information to balance work among themselves (without resorting to starting apprentices or deleting themselves[12]) as well as to order apprentices and reorder themselves after deletions. Apprentices become full squad members after a full cycle.

[12] assuming that any request may be served by any squad member, such as in random-name-generating squads and a rounding timeout squad.

Asynchronous variant

Messages are stamped with virtual ticks (which do not cycle back to 0). Each subsequent message from a member must be stamped by the next tick. A virtual-tick-0 message may be sent without restriction. After that, a member may only send a message at virtual tick v if it has received messages stamped with all virtual ticks before v. Messages may arrive out of order, but information from them should be handled in virtual tick order. When information is $\lceil \log_2 n \rceil$ virtual ticks old, it may be used as though it were known to all members.

If a member refuses to send a message because it has no new information to disseminate, it might block disseminations in progress. Each member therefore keeps track of `LastRound`, whose value is placed in each dissemination message. A member must send messages, even if they are null, until its virtual tick reaches `LastRound`. If a member generates new information at virtual tick v, it sets `LastRound` to $v + \lceil \log_2 n \rceil$. If a member receives a message containing a `LastRound` value greater than its own, it changes its own.

Using asynchronous dissemination rather than complete broadcast cuts the time for dissemination down from $O(n)$ to $O(\log n)$. If all members have information to disseminate, the number of messages is reduced from n^2 to $n\lceil \log_2 n \rceil$.

5.2. Distributed environments

We need to reevaluate the policy for adapting squads in a distributed environment. No doubt the parameters will have to be adjusted, because the cost of forwarding messages will increase. More important, we see a problem with the policy when either the client is on a lightly loaded machine and the server is on a heavily loaded machine or vice-versa. In a distributed environment, Yackos could be constructed so that, in combination with a load balancing squad, the failed message count can exert back pressure when the entire multicomputer is busy. It is not clear whether that design or another would best achieve the original goals of Yackos. Therefore, in a distributed environment, there may be a need for a squad that collects information about the "busyness" of the entire multicomputer and gives advice to squads based on this data.

6. Related work

Much work has been reported on communication kernel operating systems[3,6,11,16-18].

Medusa[19,20] provides hooks for dynamically reconfigurable task forces, which are similar to squads. Each member governs a subset of the machines; clients find the right member by inspecting a table on their machine. However, for servers like the lock server, where squad members must take responsibility for certain resources, this method does not suffice. Members of a single task force share data, and they use locks to achieve exclusive access to that data. Medusa's suggested method for policy requires kernel assistance for load measurement.

Process migration [21,22] for the purpose of load balancing[23,24] shares a goal with squads. Both deal with dynamic fluctuations in loads and service needs. Migration for load balancing responds to processor load changes; squads respond to process (server) load changes. Squad clients can generally view the creation of an apprentice or destruction of a member as a migrated server. Both migration and squad maintenance need to minimize performance penalties in the mechanism. Policy modules must stay abreast of load changes.

We pursued different directions from those usually taken in solving the migration problem. Our decisions were driven by two philosophies:

- All processes should not pay a penalty for services required by only a few. In particular, not all squads may need to be highly reliable. Similarly, not all processes require squad facilities, so to keep all message passing fast, the Yackos kernel should remain ignorant of squads.
- Make the usual case fast, but unusual cases may be slower[25]. For example, contacting a different member in the middle of a conversation is a rare event so it may be slower.

Most load balancing policies require either a supervisor or group negotiation. We avoid building supervisors or using Yackos to collect statistics or approve growth and shrinkage in squads. We choose a statistic that can be collected autonomously and inexpensively by each member.

Messages destined for an apprentice or extinct squad member must arrive at the correct member, just as messages sent to a migrating process need to be delivered to the migrated process. Usually migration

mechanisms inform clients of migrated servers of the new address. Our squads notify clients of a newly responsible member's address, but this new address is only a hint. The absolutes are kept at tables within the squad members.

For reliability, migration mechanisms tend to remove all traces of a migrating process from its source machine. If squads followed this practice, parents would destroy knowledge of their newly created apprentices. Instead, members forward messages and maintain forwarding tables to avoid extra communication with name-providing servers. We view reliability as an issue orthogonal to performance and do not treat it here except to suggest a possible solution. A squad can choose its degree of reliability by running several copies of each member in a way similar to Circus[26].

When a server is migrated, its internal data structures do not change. Creating apprentices or deleting members violates this location independence in most cases. Our squad members keep cookies, that is, names (pointers to resources) returned to clients by previously responsible squad members, in order to speed access in the usual case (the member does not give up responsibility for the resource during a conversation with a client). Newly responsible members maintain lists of old and new cookies that are used as absolutes. A failed sanity check for a presented cookie requires a search of this alias list.

ABCAST in ISIS[10,27] is intended to promote group consistency when clients may make requests of any member. Their intention was to provide reliability, but their protocols, particularly ABCAST, could be used for intra-squad communication. This broadcast method provides an alternative to the asynchronous dissemination described above, but it requires more messages and more time if many of the members have information to be broadcast.

There is support for communication to and from groups that may grow and shrink in V[28]. However, nothing has been reported about how the group decides to adapt, only what happens if a process joins or leaves a group set up for multicast purposes.

7. Experimental confirmation

We have conducted preliminary experiments to determine both that the message passer keeps up with demand and that the criteria we use for splitting are effective when the servers are busy. We counted how many times the message passer fails to pass a message because the receiver's busy input queue is full. We let the lock server automatically renew expired timeouts. This policy stresses both the lock and timeout squads heavily. The table below shows that many messages the timeout squad sends to the lock squad fail until the lock squad grows, and that the lock squad suffers extra failed messages until the timeout squad grows. These figures are the total number of messages sent in a 5-minute interval. Reducing the timeout delay showed that the message passer could easily transmit messages ten times as frequently.

members	squad	sent	failed
1	timeout	6651	10731
1	lock	3661	2182
1	timeout	5883	5850
2	lock	3391	4801
2	timeout	8508	5104
2	lock	3928	2424

An alternative to self-adjusting server classes is fixed-size server classes. In Figure 2, we compare the average response ratios of squads and various fixed-size server classes.

In this experiment there were 100 processors and 25 squads. The mean service rate was 5 seconds, and the mean rate for requests was chosen anew every 500 seconds for each server group from a uniform distribution varying between 1 and 25 seconds. The number of clients was taken to be inversely proportional to the interarrival means. The number of lightweight threads per squad member was 2; each squad member allocated input queues with capacity for three messages. For this experiment, a member adds an apprentice after seeing 5 busy signals in a row and attempts to delete itself after discovering empty queues 50 times in a row. These thresholds were multiplied by a work factor that represents the slowdown of message passing when the number of processes exceeds the number of processors. Because earlier experiments revealed that the message passing time is negligible in comparison with the service time, and to keep the simulation simple, we ignore message passing time and the time needed to fork an apprentice.

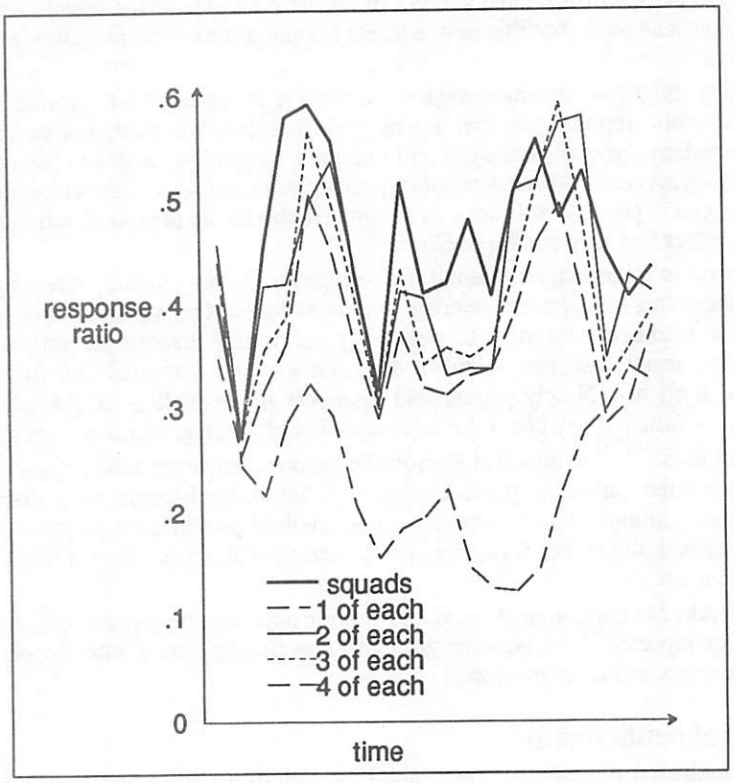


Figure 2: Static versus dynamic groups.

As expected, the worst sizes for 100 processors and 25 server classes is to have either 1 or 4 members per class; 2 or 3 members of each kind do almost as well as dynamic squads. However, should the number of total processors change, the squads would adapt and the fixed size-server classes would not.

Acknowledgements

We acknowledge support of the University of Kentucky Major Research Instrumentation Bond Program in the purchase of equipment used in this study. Tom Shelton helped implement Yackos, and Scott Dawson and Brian Sturgill provided helpful design suggestions.

References

1. Raphael Finkel and Debra Hensgen, "YACKOS on a shared memory multiprocessor," *Special Issue of Computer Architecture News on Architectural Support for Operating Systems* **16**(4)(September 1988).
2. R. A. Finkel and D. Hensgen, "YACKOS on a shared-memory multiprocessor," Technical Report 125-88, University of Kentucky Department of Computer Science (August, 1988).
3. Avadis Tevanian Jr. and Richard F. Rashid, "MACH: A Basis for Future UNIX Development," CMU-CS-87-139, Carnegie Mellon University (June 1987).
4. J.M. Barton and J.C. Wagner, "Beyond Threads: Resource sahring in UNIX," *USENIX Winter Conference*, (February 9-12, 1988).
5. M. L. Scott, "Design and implementation of a distributed systems language," Ph. D. Thesis, Technical Report #596, University of Wisconsin–Madison (May 1985).
6. A. S. Tanenbaum and S. J. Mullender, "An overview of the Amoeba distributed operating system," *ACM Operating Systems Review* **15**(3) pp. 51-64 (July 1981).
7. R. A. Finkel, "Lightweight tasks under Unix," Technical Report 103-87, University of Kentucky Department of Computer Science (December, 1987).

8. R. A. Finkel, "Tree-structured timing wheels," Technical Report 109-88, University of Kentucky Department of Computer Science (March, 1988).
9. David Cohrs, Barton P. Miller, and Lisa A. Call, "Distributed Upcalls: A Mechanism for Layering Asynchronous Abstractions," *UWM Tech Report #729*, (1987).
10. Kenneth P. Birman and Thomas A. Joseph, "Reliable Communication in the Presence of Failures," *ACM TOCS* **5** pp. 47-76 (Feb 1987).
11. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 128-139 (10-13 October 1983). In *ACM Operating Systems Review* 17:5
12. K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *SOSP*, pp. 123-138 (1987).
13. L. Lamport, "Time, clocks and the ordering of events in a distributed system," *CACM* **21**(7) pp. 558-565 (July 1978).
14. Y. Han and R. Finkel, "An optimal scheme for disseminating information," *Proceedings of the 1988 International Conference on Parallel Processing* **II** pp. 198-203 (August, 1988).
15. L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM TOPLAS*, pp. 254-280 (1984).
16. R. A. Finkel, M. L. Scott, Y. Artsy, and H.-Y. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Transactions on Software Engineering* **15**(6) pp. 676-685 (June 1989). Extended abstract presented at the *IEEE Workshop on Design Principles for Experimental Distributed Systems*, Purdue University, 15-17 October 1986
17. Y. Artsy, H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software* **4**(1) pp. 22-28 (January 1987).
18. L. D. Wittie and A. M. van Tilborg, "MICROS, a distributed operating system for MICRONET, a reconfigurable network computer," *IEEE Transactions on Computers* **C–29**(12) pp. 1133-44 (December 1980).
19. J. K. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An experiment in distributed operating system structure," *CACM* **23**(2) pp. 92-105 (February 1980).
20. John K. Ousterhout, *Medusa A Distributed Operating System*, UMI Research Press, Ann Arbor, Michigan (1981).
21. Y. Artsy and R. Finkel, "Designing process migration — The Charlotte Experience," *IEEE Technical Committee on Operating Systems Newsletter* **3**(1)(February 1989).
22. A. B. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software — Practice and Experience* **15**(9) pp. 901-913 (September 1985).
23. A. B. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," *Software —Practice and Experience* **15**(8) pp. 725-737 (August 1985).
24. P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm for a Multicomputer," Computer Sciences Technical Report #539, University of Wisconsin–Madison (April 1984).
25. Butler W. Lampson, "Hints for Computer System Design," *Proceedings of the 9th SOSP,Operating Systems Review* **17**(5) pp. 33-48 (October 1983).
26. Eric C. Cooper, "Replicated Distributed Programs," *Proceedings of the 10th SOSP*, pp. 63-78 (1985).
27. Thomas A. Joseph and Kenneth P. Birman, "Low cost management of replicated data in fault-tolerate distributed systems," *ACM Transactions on Computer Systems* **4** pp. 54-70 (February 1986).
28. David R. Cheriton and Willy Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Transactions on Computer Systems* **3** pp. 77-107 (May 1985).

Fine-Grain Scheduling

Henry Massalin and *Calton Pu*

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

We introduce the concept of *fine-grain scheduling*. Conventional scheduling makes job assignment an exclusive function of time. We broaden the meaning of the term "scheduling" to include job assignment as a function of other reference frames, such as I/O interrupts, queue overflow/underflow, and system call traps, in addition to timer interrupts. Fine-grain scheduling actions and policy adjustments (at sub-millisecond intervals), combined with a wide choice of reference frames, create adaptive, self-tuning systems.

We have implemented fine-grain scheduling in the Synthesis operating system based on a software mechanism similar to the hardware phase locked loop. Very low overhead context switches and scheduling cost (a few microseconds on a 68020-based machine) makes Synthesis fine-grain scheduling practical. Interesting applications of fine-grain scheduling include I/O device management, real-time scheduling, highly sensitive adaptive scheduling, and distributed adaptive scheduling.

1 Introduction

Traditional scheduling policies use some global property, such as job priority, to reorder the jobs in the ready queue. A scheduling algorithm is called adaptive if the global property changes dynamically, such as the total amount of CPU consumed by the job. A major problem of such global scheduling is that it assumes that all jobs are independent of each other. In a pipeline of processes, where successive stages are coupled through their input and output, this assumption does not hold. In fact, a global adaptive scheduling algorithm may lower the priority of a CPU-intensive stage, making it the bottleneck and slowing down the whole pipeline.

We call scheduling policies *fine-grain* if they take into account local information and coupling in addition to global properties. An example of interesting local information for scheduling is the size of the job's input queue: if it is empty, dispatching the job will merely block for lack of input. Fine-grain scheduling policies are sensitive to system state changes and coupling between processes. In this paper, we focus on the coupling between processes in a pipeline using the local information in the length of queues linking the processes.

Traditional scheduling mechanisms have high scheduling and dispatching overhead that discourages frequent scheduler decision making. Consequently, most scheduling algorithms tend to minimize their actions. We observe that high scheduling and dispatching overhead is a result of implementation, not an inherent property of all scheduling mechanisms. We call scheduling mechanisms *fine-grain* if their scheduling/dispatching costs are much lower than a typical CPU quantum, for example, context switches of tens of microseconds compared to CPU quanta of milliseconds.

Fine-grain scheduling policies and mechanisms together are called "fine-grain scheduling", implemented in the Synthesis operating system. Our approach to fine-grain scheduling policies is similar to feedback mechanisms in control systems. We take a job to be scheduled and measure its progress, making scheduling decisions based on the measurements. For example, if the job is "too slow", say its input queue is getting full, we schedule it more often and let it run longer.

The key idea in fine-grain scheduling policy we propose is based on feedback control, in particular phase locked loop (PLL). A hardware PLL outputs a frequency synchronized with a reference input

[0]This work is partially funded by the New York State Center for Advanced Technology on Computer and Information Systems under the grant NYSSTF CU-0112580, by the AT&T Foundation under a Special Purpose Grant, and by the National Science Foundation under the grant CDA-88-20754.

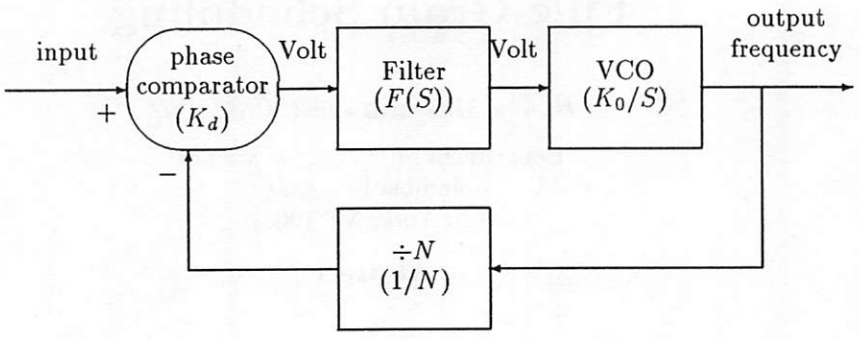


Figure 1: PLL Picture

frequency. Our software analogs of the PLL track a stream of interrupts to generate a new stable source of interrupts locked in step. The reference stream comes from a variety of sources, say an I/O device, e.g. disk index interrupts that occur once every disk revolution, or the interval timer, e.g. at the end of a CPU quantum. For readers unfamiliar with control systems, PLL is summarized in section 2.

Fine-grain scheduling would be impractical without fast interrupt processing, fast context switching, and low dispatching overhead. Interrupt handling should be fast, since it is necessary for dispatching another process. Context switch should be cheap, since it occurs often. The scheduling algorithm should be simple, since we want to avoid a lengthy search or calculations for each decision.

We have generalized scheduling from job assignments as a function of time, to job assignments as a function of any source of interrupts. We divide a job into fine-grain chunks and "schedule" them according to a reference frame, in particular timer interrupts. In fine-grain scheduling, a reference frame is simply a stream of interrupts, which can be generated by a timer, an I/O device, or even another program. This is illustrated by various applications in Synthesis, such as disk sector finding, real-time scheduling, and distributed adaptive scheduling.

2 Principles of Locked Loops

2.1 Hardware Phase Locked Loop

Figure 1 shows the PLL as a block diagram. The PLL synchronizes an internally-generated interrupt rate (output frequency) with an external interrupt rate (input frequency). If the rate divider (N) is set to unity, then the PLL generates an output that is frequency and phase synchronized to the input (frequency is the time derivative of phase). The phase detector outputs a signal proportional to the difference in phase (frequency) between its two inputs. The filter is used to tailor the time-domain response of the loop. An example is a low-pass filter that attenuates the quickly varying phase differences and passes the slowly varying phase differences. The oscillator (in hardware implemented as a voltage-controlled oscillator – VCO) generates an output frequency proportional to its input which comes from the output of the filter. The overall loop operates to compensate the variations on input, so that if the output rate is lower (higher) than the input rate, the phase detector, filter, and oscillator work together to increase (decrease) the output rate until it matches the input. When the two rates match, the output rate tracks the input rate and the loop is said to be locked to the input rate.

Our fine-grain scheduling policies have the same three elements of the PLL. First, we track the difference between the running rate of a job and the reference frame; this is analogous to a phase comparator. Second, we use a filter to dampen the oscillations in the difference, like the PLL filter. Third, we re-schedule the running job to minimize its error compared to the reference, in the same way the VCO is adjusted. Let us consider the example of a disk driver.

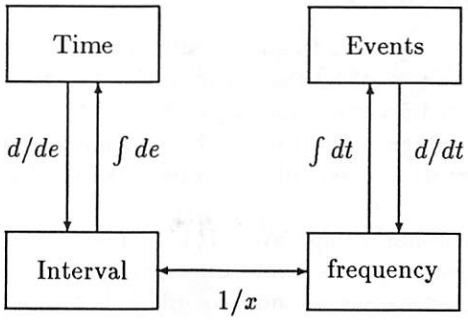


Figure 2: Relationship between ILL and FLL

A rotating disk drive generates one interrupt (sector 0) for each revolution. To read an entire track would entail an average of half-revolution rotational delay, in addition to the actual reading. With PLL-generated sector interrupts, since we know which sector is currently under the head, we can immediately read from it up to the last sector. Another read from sector 0 to the starting sector completes the track. This way we always take one rotation to read a track, regardless of where the head is. To find approximately which sector the disk head is flying over, we use a software analog of the PLL to subdivide the revolution into as many parts as there are sectors. For the disk sector interrupts, our software analog of VCO is the hardware interval timer, which generates the actual interrupts; the input reference is the disk index interrupt; the phase comparator and filter are algorithms described in section 3.2.

2.2 Software Locked Loops

When we use software to implement the PLL idea, we find more flexibility in measurement and control. Unlike hardware PLLs, where we always measure phase differences, in software we can measure either the frequency of the input (events/second), or the time interval between inputs (seconds/event). Analogously, we can adjust either the frequency of generated interrupts or the intervals between them. Combining the two kinds of measurements with the two kinds of adjustments, we get four kinds of software locked loops. In this paper, we will only look at software locked loops that measure and adjust the same variable. We call a software locked loop that measures and adjusts frequency an FLL (frequency locked loop) and a software locked loop that measures and adjusts time intervals an ILL (interval locked loop).

In general, all stable locked loops minimize the error (feedback signal). Concretely, an FLL measures frequency by counting events, so the natural behavior is to maintain the number of events (and thus the frequency) equal to the input. An ILL measures intervals, so the natural behavior is to maintain the interval between consecutive output interrupts equal to the interval between inputs.

This natural behavior can be modified with software analogs of hardware filters. The overall response of a software locked loop is determined by the kind of filter it uses to transform measurements into adjustments.[1] A low-pass filter makes the FLL output frequency and ILL output intervals more uniform. An integrator filter can handle linear increases or decreases in either frequency or interval, making both FLL and ILL more accurate and stable. A derivative filter improves response time when the input frequency or interval changes suddenly but stays with the new value. Like their hardware analogs, these filters can be combined to improve both the response time and stability of the SLL.

[1] Due to space constraints we omit the description of filters and their properties. Interested readers may consult control theory books [6].

2.3 Application Domains

At first, measuring and adjusting frequency and intervals seem equivalent, since one is the reciprocal of the other, and both kinds of feedback will work. We choose the appropriate feedback mechanism depending on the desired accuracy and application. Accuracy is an important consideration because we can only measure integer quantities: either the number of events (frequency), or the clock ticks between events (interval). We would like to measure the larger quantity of the two since it carries the higher precision.

Let us consider a scenario that favors ILL. Suppose you have a microsecond-resolution interval timer and the input event occurs about once per second. To make the output interval match the input interval, the ILL measures second-long intervals with a microsecond resolution timer, achieving 6-figure accuracy with only two events. Consequently, ILL stabilizes very quickly. In contrast, by measuring frequency (counting events), an FLL needs more events to detect and adjust the error signal. In the FLL Demo 11, it takes about 50 input events (in about 50 seconds) for the output to stabilize to within 10% of the desired value.

A second scenario favors FLL. Suppose you have an interval timer with the resolution of one-sixtieth of a second. The input event occurs 30 times a second (once every 33 milliseconds). Since the FLL is independent of timer resolution, its output will still stabilize to within 10% after seeing about 50 events (in about 1.7 seconds). However, since the event interval is comparable to the resolution of the timer, an ILL will suffer loss of accuracy. In this example, the measured interval will be either 1, 2 or 3 ticks, depending on the relative timing between the clock and input. Thus the ILL's output can have an error of as much as 50%.

Generally, slow input rates and high resolution timers would favor ILL, while high input rates and low resolution timers would favor FLL. Sometimes the problem at hand would force a particular choice. For example, in queue handling procedures, the number of get-queue operations must equal the number of put-queue operations. This forces the use of an FLL, since the actual number of events control the actions. In another example, subdivision of a time interval (like in the disk sector finder), an ILL is best.

3 Synthesis Implementation

3.1 Synthesis Operating System

Synthesis is a distributed operating system being developed by the authors at Columbia University, Department of Computer Science. A combination of high-level model of computation with high performance distinguishes Synthesis from other operating systems. The Synthesis model of computation, called *synthetic machine*, is described in a companion paper [4]. A synthetic machine has three components, units of computation (threads), units of storage (memory), and units of data movement (I/O devices). The interface to the synthetic machine is at a level comparable to that of UNIX.

To achieve high performance with a high-level interface, we use kernel code synthesis, which is described in another paper [5]. The main idea of kernel code synthesis is to generate specialized (thus short and small) code at run-time for frequently executed kernel calls. The Synthesis kernel uses three methods to optimize the code during dynamic code synthesis: Factoring Invariants to bypass redundant computations, Collapsing Layers to eliminate unnecessary procedure calls and context switches, and Executable Data Structures to shorten data structure traversal time.

The current implementation of Synthesis runs on an experimental machine (called the Quamachine), which is similar to a SUN-3: a Motorola 68020 CPU, 2.5 MB no-wait state main memory, 390 MB hard disk, $3\frac{1}{2}$ inch floppy drive. In addition, it has some unusual I/O devices: two-channel 16-bit analog output, two-channel 16-bit analog input, a compact disc (CD) player interface, and a 2Kx2Kx8-bit framebuffer with graphics co-processor.

The Quamachine is designed and instrumented to aid systems research. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and a microsecond-resolution interval timer. The CPU can operate at any clock speed from 1 MHz up to

```
operation          time (usec)
============================
create thread          142
start/stop               8
full context switch     11 (*)
FLL/ILL update step      2
Block/Unblock thread     4

(*) If the thread does not use the Floating Point co-processor.
```

Figure 3: Synthesis Fine-Grain Scheduling

```
                int residue=0, freq=0;
/*   Master (reference frame)   */     /*    Slave (derived interrupt)    */
i1()                                   i2()
{                                      {
    residue += 4;                          residue--;
    freq += residue;                       freq += residue;
        .                                      .
        .                                      <do work>
     <do work>
        .                                      next_time = NOW + 1/freq;
        .                                      schedintr(i2, next_time);
    return;                                    return;
}                                      }
```

Figure 4: Sample FLL — No Filter

50 MHz. Normally we run the Quamachine at 50 MHz. By setting the CPU speed to 16.7 MHz and introducing 1 wait-state into the memory access, the Quamachine can emulate the performance of a SUN-3/160. We validate this emulation with some CPU- and memory-intensive programs, which report the same wall-clock time for the SUN-3 and Quamachine.

Applying kernel code synthesis, the kernel call synthesized to read one character from /dev/mem takes about 15 microseconds on the Quamachine. This and other important aspects of the Synthesis kernel implementation are described in a companion paper [3]. But the key to the implementation of fine-grain scheduling mechanism in Synthesis is the extremely fast interrupt handling and context switches. Figure 3 contains measurements of Synthesis thread primitives taken from the Quamachine in the SUN-3 emulation mode. For comparison, context switches take a few hundreds of microseconds in a high performance real-time operating system [2].

3.2 FLL Examples

Applying the feedback idea to scheduling, we use the FLL mechanism to keep two processes or interrupt sources running at some algebraic function of each other. Figure 4 shows the general abstract algorithm (without filters) when one source of interrupts happens at 4 times the rate of the other. The algorithms described in figures 4, 5, 6, and 7 describe filters to improve the responsiveness and stability of the FLL. Each one of them has a feedback system analog [6]. All the sample FLL algorithms shown in this section are meant to illustrate the FLL mechanism; they are not actual Synthesis code. Please see appendix A for examples of working code for IBM PC/AT class of machines. Figure 6 contains line numbers (1.1, 2.1, etc.) that will be used in appendix A.

The algorithm in figure 4 represents a simple FLL. The phase variable, `residue`, keeps track of relative rates of `i1` and `i2`. The variable `freq` holds the frequency of i2 interrupts, and `1/freq` the time between successive `i2` interrupts. `Freq` has `residue` added to it each time `i1` or `i2` executes. Summing `residue` to `freq` corresponds to the hardware PLL's implicit integrator inside the VCO. The process `i1` is the reference; it runs at its own rate; and each time it executes it adds 4 to the `residue` counter. Process `i2` runs at 4 times the rate `i1` runs and each time `i2` executes it

```
int residue=0, freq=0, lopass=0;

i1()
{
    residue += 4;
    lopass = (7*lopass + residue)/8;
    freq += lopass;
        .
        .
     <do work>
        .
        .
    return;
}
```

```
i2()
{
     residue--;
     lopass = (7*lopass + residue)/8;
     freq += lopass;
         .
      <do work>
         .
     next_time = NOW + 1/freq;
     schedintr(i2, next_time);
     return;
}
```

Figure 5: Sample FLL — Low-pass Filter

```
int residue=0, freq=0, lopass=0, old_r=0;

i1()
{
1.1 residue += 4;
1.2 lopass = (7*lopass + residue)/8;
1.3 freq += lopass + (residue - old_r);
1.4 old_r = residue;
        .
        .
     <do work>
        .
        .
    return;
}
```

```
i2()
{
2.1  residue--;
2.2  lopass = (7*lopass + residue)/8;
2.3  freq += lopass;
         .
       <do work>
         .
2.4  next_time = NOW + 1/freq;
2.5  schedintr(i2, next_time);
     return;
}
```

Figure 6: Sample FLL — Derivative and Low-pass Filter

```
int residue=0, freq=0, integral=0;

i1()
{
    residue += 4;
    integral += residue;
    freq += integral;
        .
        .
     <do work>
        .
        .
    return;
}
```

```
i2()
{
     residue--;
     integral += residue;
     freq += integral;
         .
      <do work>
         .
     next_time = NOW + 1/freq;
     schedintr(i2, next_time);
     return;
}
```

Figure 7: Sample FLL — Integral Filter

decrements the `residue` counter. If `i2` and `i1` were running at the perfect relative rate of 4 to 1, `residue` would tend to zero and no correction would result. In contrast, if `i2` is slower than 4 times `i1`, residue will become positive, increasing the frequency of `i2` interrupts and causing `i2` to speed up. Similarly, if `i2` is faster than 4 times `i1`, `i2` will be slowed down. As the difference in relative speeds increases, the correction gets correspondingly larger. As `i1` and `i2` approach the exact rate of 1:4, the difference decreases and we reach the minimum correction with `residue` being decremented by one and incremented by four, therefore cycling between $[-2, +2]$. A non-zero `residue` will cause the `i2` execution frequency to jitter, even though `i1` and `i2` were close to the ideal execution rate.

A low-pass filter in the program helps eliminate this jitter at the expense of settling time. Figure 5 shows an FLL with low-pass filter. The variable `lopass` keeps a "history" of what the most recent `residues` were. Each update adds 1/8 of the new `residue` to 7/8 of the old `lopass`. This has the effect of taking a weighted average of recent `residues`. When residue is positive for many iterations, as is the case when `i2` is too slow, `lopass` will eventually be equal to `residue`. But if residue oscillates, as in the situation described in the previous paragraph, `lopass` will go to zero.

The problem now is increased settling time. The low-pass filter has a lag effect on the FLL response. If `i1` speeds up quickly, `i2` will lag behind `i1` while `lopass` "charges up". Settling time can be decreased by adding a differentiator filter in the `i1` loop (figure 6). The expression `(residue - old_r)` approximates the first derivative of residue. Since it appears only in the `i1` loop, it does not magnify the high frequency `i2` jitter. The correction due to derivative is higher when `i1` execution rate varies quickly, pushing `i2` towards the right rate quickly.

Finally, we show an integrator filter in figure 7. This kind of filter is useful for accurate tracking of interrupt sources with linearly increasing (or decreasing) rates. Since integrals filter out high frequencies naturally, there is less need for a low-pass filter to deal with jitter.

3.3 Synthesis Examples

We have used the locked loop scheduling policies to handle a wide variety of jobs in Synthesis. These are:

- An ILL rhythm tracker for a special effects sound processing program.
- An ILL that adjusts itself to the disk rotation rate, generating an interrupt a few microseconds before each sector passes under the disk head.
- A digital oversampling filter for a CD player. An FLL is used to adjust the filter I/O rate to match the CD player.

The special effects program takes as input a stereo sound source (at 44,100 Hertz), which is digitized from a microphone or other analog sound sources, or a direct digital CD player output. Then the program processes the input in real time and produces output which is sent to the digital to analog converters and eventually to the speakers. The processing includes delay elements, echo and reverberation filters, adjustable low-pass, band-pass and high-pass filters, and a correlator and feature extraction unit that can drive the other stages of processing. We use the correlator to extract rhythm pulses from the music. These are fed to an ILL, which generates interrupts synchronized to the beat of the music. These interrupts are then used to add more drum beats to the music, or to substitute a new rhythm track. You can also get pretty pictures synchronized to the music when you plot the ILL input versus output on a graphics display.

There is an FLL at work here as well. It is the system-wide scheduler FLL that keeps the I/O queues from overflowing. The CD player driver relies on this feature. In Synthesis, reading from the CD player is no different than reading from any other device or file. Simply open `"/dev/cd"` and read from it. To listen to the CD player, one could use the program in figure 8. The scheduler FLL keeps the data flowing smoothly at the 44.1 KHz sampling rate, regardless of how many CPU-intensive jobs might be executing in the background.

An ILL helps the disk driver minimize rotational delay by generating sector interrupts. The disk controller generates an interrupt every disk revolution. The ILL synchronizes to this interrupt, and creates a new source of interrupts corresponding to each sector. Thus the disk driver knows what

```
main()
{
        char    buf[100];
        int     n, fd1, fd2;

        fd1 = open("/dev/cd", 0);
        fd2 = open("/dev/speaker", 1);
        for(;;) {
                n = read(fd1, buf, 100);
                write(fd2, buf, n);
        }
}
```

Figure 8: Program to Play a CD

sectors are closest to the disk heads and can perform rotational optimization in addition to normal seek optimization algorithms.

The FLL is used in the digital interpolator filter for the CD player. A digital interpolator takes as input a stream of sampled data and creates additional samples in-between the original ones by interpolation. This oversampling increases the accuracy of analog reconstruction of digital signals. We use 4:1 oversampling, i.e. we generate 4 samples using interpolation from each CD sample. The CD player has a new data sample available 44,100 times per second, or one every 22.68 microseconds. The interpolated data output is four times this rate, or one every 5.67 microseconds.[2] We use an FLL to generate an interrupt source at this rate, synchronized with the CD player.

3.4 Discussion

A formal analysis of fine-grain scheduling is beyond the scope of this paper. However, we would like to give the readers an intuitive feeling about two situations: saturation and cheating. As the CPU becomes saturated (no idle times), the PLL-based scheduler degrades gracefully. The processes closest to externally generated interrupts (device drivers) will still get the necessary CPU time. The CPU-intensive processes away from I/O interrupts will slow down first, as they should at saturation.

Another potential problem is cheating (consuming resources unnecessarily to increase priority), since fine-grain scheduling tends to give more CPU to processes that consume more. However, cheating cannot be done easily from within a thread or by cooperation of several threads. First, unnecessary loops within a program does not help the cheater, since they do not speed up data flow in the pipeline of processes. Second, I/O within a group of threads only shifts CPU quanta within the group. A thread that reads from itself gains quantum for input, but loses the exact amount in the self-generated output. To increase the priority of a process, it must read from a real input device, such as the CD player. In this case, it is virtually impossible for the OS kernel to distinguish the real I/O from cheating I/O. This kind of cheating would succeed under existing schedulers.

4 Applications

We apply fine-grain scheduling policies to three kinds of situations:

- interrupt source coupled to interrupt source, described in section 4.1,
- interrupt source coupled to program progress, described in section 4.2, and
- program progress coupled to program progress, described in sections 4.3 and 4.4.

[2] This program runs on the experimental machine at 50 MHz clock rate.

4.1 Interrupts with Integral Stability

The FLL provides integral stability, i.e. the long-term drift between the reference frame and generated interrupts tend to zero, even though any individual interval may differ from the reference. This is in contrast with differential stability, in which consecutive intervals are the same, but any systematic error, no matter how small, will accumulate into a long-term drift. This is one of the reasons why a sufficiently accurate interval timer can generate a stream of interrupts with good differential stability but not integral stability.

We would like to obtain integral stability when we wish to synchronize a stream of interrupts with respect to an external interrupt source: a precise atomic clock, the CD player, analog to digital coverters, etc. In these cases, small discrepancies are acceptable as long as they do not accumulate into a long-term drift. Using PLL, the input is the external interrupt source. The output is a new stream of interrupts occurring at some rational (p/q) rate of the input. The PLL adjusts an interval timer so that each interrupt occurs as close to the "correct" time of arrival as possible given the resolution of the interval timer, while maintaining integral stability – p interrupts out for every q interrupts in. One use of this mechanism is the CD player oversampling interpolator described in section 3.3.

4.2 Real-Time Scheduling

We divide the hard-deadline jobs into two categories: the short ones and the long ones. A short job is one that must be completed in a time frame within an order of magnitude of interrupt and context switch overhead. For example, a job taking up to 50 microseconds would be a short job in Synthesis. Short jobs are scheduled as they arrive and run to completion without preemption.

Long jobs take longer than 100 times the overhead of an interrupt and context switch. In Synthesis this includes all the jobs that take more than 1 millisecond, which includes most of the practical applications. The main problem with long jobs is the variance they introduce into scheduling. If we always take the worst scenario, the resulting hardware requirement is usually very expensive and unused most of the time.

To use fine-grain scheduling policies for long jobs, we break down the long job into small *strips*. For simplicity of analysis we assume each strip to have the same execution time ET. We define the estimated CPU power to finish job J as:

$$Estimate(J) = \frac{(\text{strips in J}) * ET}{Deadline(J) - Now}$$

For a long job, it is not necessary to know ET exactly since the locked loop "measures" it and continually adjusts the schedule in lock step with the actual execution time. In particular, if $Estimate(J) > 1$ then we know from the current estimate that J will not make the deadline. If we have two jobs, A and B, with $Estimate(A) + Estimate(B) > 1$ then we may want to consider aborting the less important one and calling an short emergency routine to recover.

Unlike traditional hard-deadline scheduling algorithms, which either guarantee completion or nothing, fine-grain scheduling provides the ability to predict the deadline miss. We think this is an important practical concern to real-time application programmers, especially in recovery from faults. (For a good discussion of issues in real-time computing, see [7].)

4.3 Adaptive Scheduling

Current operating systems, e.g. UNIX, use an adaptive strategy to improve system throughput. Usually, they have a priority-based scheduling mechanism. CPU-intensive jobs have their priority lowered and IO-intensive jobs priority increased. In UNIX, if a job has exhausted its CPU quantum when de-scheduled then it is likely to be CPU-intensive. Also, if a job accumulated enough CPU minutes it is automatically demoted to a lower priority. This scheme works very well with jobs of long duration, since the scheduling decision is based on the average behavior of the job.

In Synthesis, we improve the adaptive scheduling further with fine-grain scheduling. Let us take the simple case of a pipeline of processes each with one input and output. Each stage of the

pipeline is a consumer of data from the previous stage and producer of data for the next stage. If one particular stage in the pipeline needs a relatively large amount of CPU (compared to the other stages), the above simple adaptive scheduling would lower its priority, causing congestion to form at the stage. In a world of many light-weight processes connected in a graph, such as the Synthesis threads, avoiding this kind of congestion is crucial for good performance.

A smooth flow of data through the pipeline would best use the resources of the system, since each stage will be running at just the right speed, without idling or congestion. In the above scenario, data flow would slow down at the CPU-intensive stage and the entire pipeline runs at the speed of the lowest priority stage. To solve this problem with a fine-grain scheduling policy, we adjust the process priority according to the length of its input queue. The frequency and length of the CPU quantum of a stage is directly proportional to its input queue and inversely proportional to its output queue. If the input queue is full, then this stage is a bottleneck and should be scheduled more often and with a larger quantum of CPU, in the hope that it will start consuming its input faster. Similarly, if a process has filled its output queue, then it is "too fast" and should have a smaller quantum and be scheduled less often.

In the Synthesis adaptive algorithm, the kernel detects when a queue is empty or full (analogous to phase comparator) and makes the adjustments to the scheduling timer (like a VCO). So a CPU-intensive process in a pipeline will be scheduled more often and "run faster" to keep up with the rest of the pipeline. This policy may seem non-intuitive, since we usually lower the priority of CPU-intensive jobs. The explanation is two-fold. First, we want to slowdown upstream jobs, since they will block often due to full queues, costing kernel call overhead. Second, we want to speed up the downstream jobs so they can use the resources (e.g. I/O devices) allocated to them more effectively. For example, getting a smooth stream of disk requests to the disk driver will result in more overlap between disk activity and CPU computations, increasing overall throughput. We should note that Synthesis does not currently do any significant global CPU accounting. With only local adjustments based on queue length, fine-grain scheduling policies exhibits global stability, i.e. the pipeline of processes run smoothly.

Since the Synthesis adaptive algorithm varies process priority dynamically, we need to put a limit on the CPU allocated to each process to avoid monopoly. For example, a process that reads from a high data rate I/O source (say a sound digitizer at 50,000 samples per second) may be able to capture a lot of CPU because the digitized sound data come in at very high rate. We impose an upper limit on the process CPU quantum and scheduling frequency to prevent any process from monopolizing CPU. This restriction may be relaxed for dedicated real-time systems.

4.4 Multiprocessor and Distributed Scheduling

We think the adaptiveness of FLL promises good results in multiprocessor and distributed systems. At the risk of oversimplification, we describe an example with fixed buffer size and execution time. We recognize that given a load we can always find the optimal scheduling statically by calculating the best buffer size and CPU quantum. We emphasize the main advantage of locked loops: the ability to dynamically self-adjust towards the best buffer size and CPU quantum. This is important when we have a variable system load, jobs with with variable demands, or a reconfigurable system with a variable number of CPUs.

Figure 9 shows the static scheduling for a two-processor shared-memory system with a common disk (transfer rate of 2 MByte/second). We assume that both processes access the disk drive at the full transfer rate, e.g. reading and writing entire tracks. Process 1 runs on processor 1 (P1) and process 2 runs on processor 2 (P2). Process 1 reads 100 KByte from the disk into a buffer, takes 100 msec to process them, and writes 100 KByte through a pipe into process 2. Process 2 reads 100 KByte from the pipe, takes another 100 msec to process them, and writes 100 KByte out to disk. In the figure, process 1 starts to read at time 0. All disk activities appear in the bottom row, P1 and P2 show the processor usage, and shaded quadrangles show idle time.

Figure 10 shows the fine-grain scheduling mechanism (using FLL) for the same system. We assume that process 1 starts by filling its 100 Kbyte buffer, but soon after it starts to write to the output pipe, process 2 starts. Both processes run to exhaust the buffer, when process 1 will read

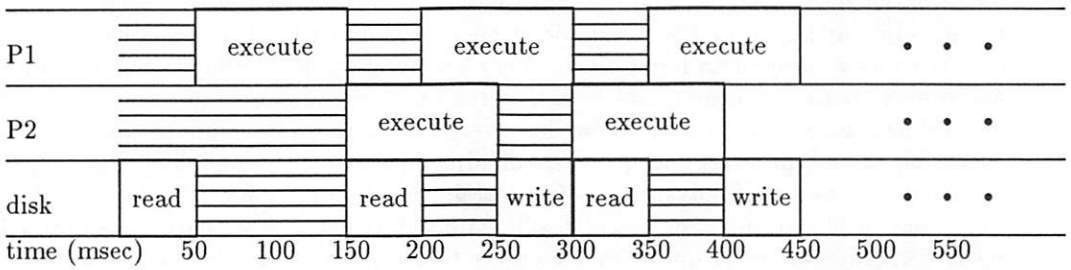


Figure 9: Two Processors, Static Scheduling

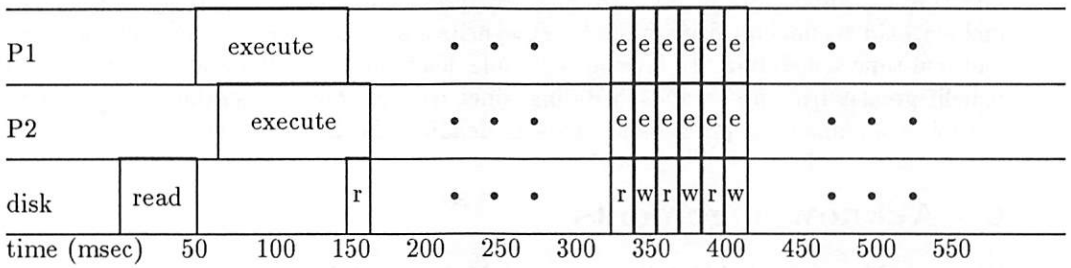


Figure 10: Two Processors, Fine-Grain Scheduling

from the disk again. After some settling time, depending on the filter used in the locked loop, the stable situation is for the disk to remain continuously active, alternatively reading into process 1 and writing from process 2. Both processes will also run continuously, with the smallest buffer that maintains the nominal transfer rate.

The above example illustrates the benefits of fine-grain scheduling policies in parallel processing. In a distributed environment, the analysis is more complicated due to network message overhead and variance. In those situations, calculating statically the optimal scheduling becomes increasingly difficult. We expect the fine-grain scheduling to show increasing usefulness as it adapts to an increasingly complicated environment.

Another application of FLL to distributed systems is clock synchronization. Given some precise external clocks, we would like to synchronize the rest of machines with the reference clocks. Many algorithms have been published, including a recent probabilistic algorithm by Christian [1]. Instead of specialized algorithms, we use an FLL to synchronize clocks, where the external clock is the reference frame, the message delays introduce the jitter in the input, and we need to find the right combination of filters to adapt the output to the varying message delays. Since an FLL exhibits integral stability, the clocks will tend to synchronize with the reference once they stabilize. We are currently collecting data on the typical message delay distributions and finding the appropriate filters for them.

5 Conclusion

We have generalized scheduling from job assignments as a function of time, to job assignments as a function of any source of interrupts. The generalized scheduling is most useful when we have fine-grain scheduling, that uses frequent state checks and dispatching actions to adapt quickly to system changes. Relevant new applications of the (generalized) fine-grain scheduling include I/O device management, such as a disk sector interrupt source, and adaptive scheduling, such as real-time scheduling and distributed scheduling.

Our implementation of fine-grain scheduling in the Synthesis distributed operating system is based on feedback systems, in particular phase locked loop. Synthesis fine-grain scheduling policy means adjustments every few hundreds of microseconds on local information, such as the number of characters waiting in an input queue. Very low overhead scheduling (a few tens of microseconds) and context switch for dispatching (less then ten microseconds) form the foundation of our fine-grain scheduling mechanism. In addition, we have very low overhead interrupt processing to allow frequent checks on the job progress and quick, small adjustments to the scheduling policy.

There are two main advantages of fine-grain scheduling: quick adjustment to changing situations, and early warning of potential deadline misses. Quick adjustments make better use of system resources, since we avoid queue/buffer overflow and other mismatches between the old scheduling policy and the new situation. Early warning of deadline misses allows real-time application programmers to anticipate a disaster and attempt an emergency recovery before the disaster strikes.

We have only started exploring the many possibilities that generalized fine-grain scheduling offers. Distributed applications stand to benefit from the locked loops, since they can track the input interrupt stream despite jitters introduced by message delays. Concrete applications we are studying include load balancing, distributed clock synchronization, smart caching in memory management and real-time scheduling. To give one example, load balancing in a real-time distributed system can benefit greatly from fine-grain scheduling, since we can detect potential deadline misses in advance; if a job is making poor progress towards its deadline locally, it is a good candidate for migration.

6 Acknowledgements

We would like to thank AMD, Hitachi, Intel, Motorola, National Semiconductor and Mupac for their contributions of hardware samples for the experimental machine. Students in the Advanced Operating Systems class at Columbia, Mei-Chun Hsu and Tom Maglione made constructive comments that improved the presentation of the paper.

References

[1] F. Christian.
Probabilistic clock synchronization.
Technical Report RJ6432 (62550) Computer Science, IBM Almaden Research Center, September 1988.

[2] S. Karsten, T. Bihari, B.W. Weide, and G. Taulbee.
High-performance operating system primitives for robotics and real-time control systems.
ACM Transactions on Computer Systems, 5(3):189–231, August 1987.

[3] H. Massalin and C. Pu.
Threads and input/output in the Synthesis kernel.
In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Arizona, December 1989.

[4] C. Pu and H. Massalin.
Model of computation in Synthesis.
Technical Report CUCS-383-88, Department of Computer Science, Columbia University, 1989.
In preparation.

[5] C. Pu, H. Massalin, and J. Ioannidis.
The Synthesis kernel.
Computing Systems, 1(1):11–32, Winter 1988.

[6] Ulrich L. Rohde.
Digital PLL Frequency Synthesizers: Theory and Design.
Prentice-Hall, Inc., first edition, 1983.

[7] J.A. Stankovic.
Misconceptions about real-time computing: A serious problem for next-generation systems.
IEEE Computer, 21(10):10–19, October 1988.

A Fun Demo for Your PC

Readers who want to play with the locked loops can run the following programs (figures 11 and 12), written for the IBM PC and compatibles. Type it and compile! It works with the Microsoft C compiler. You may have to change the `define getkey()` on other compilers. The function `getkey()` returns the key pushed and -1 if no key has been pushd (non-blocking read of keyboard). Run it. Push and hold down the "1" key, letting auto-repeat generate a steady stream of "events" and watch the error oscillate a few times to stabilize towards zero. Try playing with the parameters, change the filters, the gain constant, ... Enjoy!

The FLL demo program is patterned after the template program in figure 6 (FLL – Derivative and Low-Pass Filter). All the line references point to that figure, including the line numbers and specific statements.

- The variable `time` keeps track of simulated time, which is incremented by one each loop.
- The FLL synchronizes to a multiple of an external event, in this case the pushing of a key. The multiple is determined by the numeric key being pushed ($1 = \frac{1}{2}, 2 = 1, \ldots, 9 = 4\frac{1}{2}$) [Key = multiple].
- The output is two rotating bars, the first rotating with each key push, the second rotating at the FLL output frequency. The other numbers displayed show the current error and instantaneous frequency.
- The statement `err += 3 * (c-'0');` corresponds to line 1.1.
- The statement `if(time > next)` corresponds to the interval timer expiring and causing an interrupt.
- The statement `err -= 6` corresponds to line 2.1.
- The statement `x = filter(err)` corresponds to line 2.2.
- The statement `freq += x` corresponds to line 2.3.
- The statement `next += 200000L/freq` corresponds to the line `schedintr(...)` in figure 6.

```
/* an FLL */
#include <stdio.h>
#define ESC     27
#define getkey()        (kbhit() ? getch() : -1)

filter(x)
int     x;
{
static  int     lopass, old_x;
        int     r;

        lopass = (3*lopass + x) >> 2;
        r = lopass + 15*(x - old_x);
        old_x = x;
        return r;
}

main()
{
static  char    bar[] = "|/-\\";
        int     i1, i2, event, freq, err, x, c;
        long    time, next, last, tmp;

        i1 = i2 = err = event = 0;
        time = next = last = 0;
        freq = 200;
        while((c = getkey()) != ESC)
        {
                time++;

                /* this is i1 */
                if(c >= '1' && c <= '9') {
                        err += 3 * (c-'0');
                        x = filter(err);
                        freq += x; if(freq <= 0) freq = 1;
                        next = last + 200000L/freq;
                        i1 = (i1+1) & 3;
                        event = 1;
                }

                /* this is i2 */
                if(time > next) {
                        last = next;
                        err -= 6;
                        x = filter(err);
                        freq += x; if(freq <= 0) freq = 1;
                        next = last + 200000L/freq;
                        i2 = (i2+1) & 3;
                        event = 1;
                }

                if(event) {
                        event = 0;
                        printf("%c %c %+4.3d %+4.3d %5d\r", bar[i1], bar[i2], err, x, freq);
                }
        }
}
```

Figure 11: FLL Demo Program

```
/* an ILL */
#include <stdio.h>
#define ESC     27
#define getkey()        (kbhit() ? getch() : -1)

long    filter(x)
long    x;
{
static  int     lopass;
        x <<= 8;
        lopass = (63*lopass + x) >> 6;
        return (lopass+128)>>8;

}

main()
{
static  char    bar[] = "|/-\\";
        int     i1, i2, event, c, mul;
        long    time, next, last1, last2, intv1, intv2, err, x;

        time = next = last1 = last2 = intv1 = intv2 = 0;
        i1 = i2 = event = 0;
        err = 0;
        mul = 2;
        while((c = getkey()) != ESC)
        {
                time++;

                /* this is i1 */
                if(c >= '1' && c <= '9') {
                        mul = (c-'0');
                        intv1 = time - last1;
                        last1 = time;
                        err = ((intv1<<1)/mul) - intv2;
                        x = filter(err);
                        intv2 += x; if(intv2 <= 0) intv2 = 0;
                        next = last2 + intv2;
                        i1 = (i1+1) & 3;
                        event = 1;
                }

                /* this is i2 */
                if(time > next) {
                        if(time - last1 > intv1) intv1 = time - last1;
                        err = ((intv1<<1)/mul) - intv2;
                        x = filter(err);
                        intv2 += x; if(intv2 <= 0) intv2 = 0;
                        last2 = next;
                        next = last2 + intv2;
                        i2 = (i2+1) & 3;
                        event = 1;
                }

                if(event) {
                        event = 0;
                        printf("%c %c %9ld %9ld\r", bar[i1], bar[i2], intv1, intv2);
                }
        }
}
```

Figure 12: ILL Demo Program

The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis *

Joseph Boykin
boykin@encore.com

Alan Langerman
alan@encore.com

Mach Operating System Project
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, MA 01752-3004

Abstract

Mach is a new operating system targeted for distributed and multiprocessor environments. Mach contains 4.3BSD compatibility code that, unlike the Mach kernel, runs only on a single processor, thus presenting a performance bottleneck to the rest of the system. Pieces of the 4.3BSD compatibility code were selectively parallelized to reduce this bottleneck. Significantly improved multiprocessor and multi-user performance was achieved using minimum modification of existing data structures and algorithms. A framework was left in place for future parallelization enhancements.

1 Introduction

The Mach operating system, developed at Carnegie-Mellon University, targets a broad range of computer architectures, including uniprocessor, multiprocessor and distributed systems. The designers of Mach intend to produce a compact, efficient kernel on top of which may be layered interfaces for traditional operating systems such as 4.3BSD, System V, MSDOS, VMS, *etc.* Most traditional kernel support, such as device drivers and filesystem handling, will be provided by a set of user-level servers. The Mach kernel will provide the mechanisms necessary for simple operation in a distributed environment using uniprocessor or multiprocessor systems. Mach currently provides full backward compatibility with 4.3BSD.

Encore is interested in Mach because of its multiprocessor support. In particular, Encore is developing a DARPA-sponsored 1,000 MIPS multiprocessor that will use Mach. Encore currently runs Mach on the Multimax, a symmetric shared memory multiprocessor using the National Semiconductor 32000 family of processors.

Mach used the original 4.3BSD code in order to insure BSD compatibility. As currently distributed by CMU, Mach's 4.3BSD compatibility code has not been modified to support efficient

*This research was supported in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Multimax, UMAX4.3 and UMAXV are trademarks of Encore Computer Corporation. Unix is a trademark of AT&T Bell Laboratories. MSDOS is a trademark of Microsoft Corporation. VMS is a trademark of Digital Equipment Corporation. Neal Nelson Business Benchmark is a trademark of Neal Nelson and Associates.

multiprocessor operation. The original 4.3BSD kernel was designed for a uniprocessor: kernel data structures are protected from interrupt-level races by disabling interrupts at appropriate times. This approach does not suffice in a multiprocessor environment.

The Mach kernel is designed and implemented to execute correctly on a multiprocessor. Mach uses multiprocessor locks to synchronize operations between separate processors. These locks include spin locks (called *simple_locks*) for non-blocking synchronization and read/write locks that may cause a thread to sleep until the lock becomes available. Mutual exclusion locks are built from read/write locks. *Simple_locks* may also be used to synchronize between processors and I/O devices that operate out of main memory.

Mach resolves the contradiction between the native, inherently parallelized Mach code and the inherently serial 4.3BSD compatibility code by forcing all 4.3BSD code to execute on a single processor, the so-called *master*. We use the term *unix_master* to denote this restriction because the internal Mach function *unix_master*() ensures that a Mach thread executes on the *master* processor. Device interrupt handling is also confined to the *master* processor. Thus, the normal 4.3BSD mutual exclusion mechanisms continue to operate as expected. Obviously, any Mach code that manipulates 4.3BSD state must also be restricted to the *master* processor.

The *master* processor design works well: all user-level code and all native Mach operations (*e.g.*, Mach kernel calls, virtual memory handling and Mach IPC) execute on any available CPU. Only 4.3BSD-specific routines and the Mach code that interfaces directly to them must obey the *master* processor restriction. Ultimately the 4.3BSD compatibility code will migrate into user-level servers and become executable by any processor.

In the meantime, unfortunately, the *master* processor restriction has severe implications for overall multiprocessor performance. We observed that apparent Mach performance was significantly worse than that offered by the other Encore operating systems, UMAX4.3 (based on 4.3BSD) and UMAXV (based on System V). Even though the basic Mach functionality had been written from scratch for multiprocessor operation, the vast bulk of user code makes heavy use of the 4.3BSD compatibility code. It became clear that the 4.3BSD routines had to be modified to provide better performance.

We realized that the *unix_master* restriction offered us the opportunity to parallelize the 4.3BSD compatibility code selectively. Rather than alter all of the 4.3BSD code at one time, we could modify a piece at a time for multiprocessor operation and examine the results.

We adopted these goals:

1. minimize modifications to existing code.
2. provide a framework for future performance enhancements.
3. achieve significant performance increase with minimum work.

We pursued the most multiprocessor performance with the least effort. In effect, we followed a "90/10" rule: try to capture 90% of the possible performance improvement at a cost of 10% of the total work. (We didn't take this maxim literally, of course.) Because of our resource limitations, we preferred to implement a framework for future parallelization and tuning efforts rather than attempt to parallelize all subsystems immediately or try to implement highly parallel subsystems from scratch.

After analyzing system call counts and interrupt handling, it became clear that the greatest performance wins were to be found by parallelizing the low-level interrupt handling, the filesystem, and the network code. In general, we sought to parallelize code by adding synchronization mechanisms to existing data structures and adding appropriate calls to synchronization routines from existing algorithms. In other words, minimum modification was a cardinal rule.

The minimum modification rule was also important because we have to track functional modifications and bug-fixes to this code by Berkeley, CMU, and other organizations.

While a significant amount of work has already been done in the area of multiprocessor Unix operating systems[1, 7, 3, 2], we are unaware of any design that incorporates an incremental approach to parallelization and that attempts to achieve substantial parallelism without altering data structures or rewriting algorithms. There is certainly no other implementation that must reconcile these goals within the context of an operating system that is highly parallel in some parts but uses a master/slave relationship for the rest of the code[5].

We will describe some of the design decisions we made and implementation problems we encountered during the parallelization effort. First, we will focus on converting interrupt-level synchronization problems into multiprocessor synchronization problems. Next, we will discuss our modifications to the 4.3BSD filesystem and network code. We will also discuss our approach to debugging and statistics gathering. Finally, we will summarize our results and mention possibilities for future work.

We assume that the reader is familiar with the internals of the 4.3BSD kernel, particularly the filesystem and network code. The reader should also be aware that Mach uses tasks and threads, not Unix processes, and throughout this paper we will use the Mach terminology. The original Encore Mach port, with no modification of the 4.3BSD compatibility code, was known as Encore Mach/0.2 and derived from CMU's Release 2.0 of Mach. The current release of Encore's Mach, including the parallelized 4.3BSD code, is known as Mach/0.5.

2 Interrupt Handling

A consequence of the Mach *unix_master* design is the restriction of all interrupt handling to the *master* processor. The same processor that executes the 4.3BSD code must also execute the interrupt handling code or the 4.3BSD programming model will break. This I/O restriction is doubly ironic in our symmetric multiprocessor as other processors capable of handling the interrupts go idle while the load on the *master* processor increases.

The parallelization of both filesystem and network further demanded that interrupt handling be "fixed" because the 4.3BSD-style interrupt handling would not function with lock-based filesystem and network code. Left untouched, interrupt-level operations could attempt to take blocking locks with disastrous results.

We defined three somewhat conflicting goals for upgrading the 4.3BSD interrupt model for our multiprocessor environment:

1. Minimize work done at interrupt-level.
2. Transform interrupt-level synchronization problems into thread context synchronization problems (so multiprocessor locks could be used).
3. Avoid lengthy processing delays, where possible.

We chose to define new kernel threads that would be responsible for handling incoming interrupts. The interrupt handler would be responsible for saving appropriate information and then waking up the appropriate thread to complete the processing. For example, the Multimax has four main interrupt sources: per-processor time-slice end counters; the System Control Card (SCC) which, among other things, provides serial ports for local and remote consoles; the masstore (disk/tape) interface; and the Ethernet interface. Time-slice end activities are already handled by the Mach kernel and therefore required no additional work on our part.

2.1 Console TTY handling

The interrupt handler for the directly-connected serial ports required some recoding. Originally, the SCC interrupt handler, **slcintr**, would directly invoke SCC tty routines. In our parallelized code, however, the SCC tty routines must acquire a blocking *tty_lock* before manipulating tty data structures. We modified **slcintr** to catch the interrupt, enqueue a unit identifier on the *scc_pend_intrs* queue, then awaken the *slcintr_thread*. The *slcintr_thread* handles the normal character processing, including calling into the SCC tty routines. In this particular case, keeping up with console input is not difficult and we don't mind a delay between receiving the character and processing it so the *slcintr_thread* has a relatively low priority.

2.2 Masstore Interrupts

We have paid more attention to optimizing the handling of masstore interrupts because they are so frequent and important. A masstore interrupt signals the completion of an I/O command or the generation of an error message. **Msintr**, the masstore interrupt handler, reads, logs and discards error messages. This behavior need not change for parallelized interrupt handling. However, on an I/O completion, there may be a need to manipulate the buffer on which the I/O finished. The non-parallelized **msintr** always called into a buffer cache routine, **iodone**, to pass on the news of the I/O completion. **Iodone** might then call **brelse** to release the buffer back to the buffer cache. All of these activities took place at interrupt-level. In our parallelized filesystem, however, blocking locks synchronize access in the buffer cache. It is an error for the interrupt-level code to manipulate blocking locks.

The solution to this problem was to invent the *biodone_thread* to process all I/O completions. **Msintr** queues information about the I/O completion to the *biodone_thread*, which wakes up and calls **iodone**. Blocking locks can then be acquired in thread context.

However, the *biodone_thread* itself can become a bottleneck in the disk subsystem; there is only one thread and there is also a rescheduling delay when the thread is awakened. Furthermore, the thread will be used frequently, stealing time from other running threads. To alleviate these problems, we optimized the frequent case of a synchronous I/O completion to avoid using a *biodone_thread* at all. Normally, for a synchronous I/O, **iodone** merely has to wake up the user thread waiting for the I/O to complete; no buffer cache manipulation is needed. Therefore, we employed an "event" mechanism that allows us to post the news of a synchronous I/O completion directly from interrupt-level, awakening the sleeping thread without using the *biodone_thread* or **iodone**. (Asynchronous completions, which manipulate buffer cache state, continue to require the *biodone_thread* and **iodone**.) This optimization substantially reduces the need for the *biodone_thread*. The design and implementation permit multiple *biodone_threads* to be started in case a single *biodone_thread* becomes a bottleneck. Statistics to date suggest that a single *biodone_thread* is adequate.

2.3 Ethernet Interrupts

Interrupts from the Ethernet interface result from incoming packets, completions for outgoing packets, and error conditions. The latter two conditions are easy to handle and were already correctly implemented for multiprocessor operation. The most important matter is handling incoming packets.

It should be no surprise that the original code would not work in a multiprocessor environment. The original algorithms would process packets and massage protocol information from the network interface all the way up to the socket layer while operating the whole time at interrupt level. This design was changed to minimize the work done at interrupt-level and because operations at interrupt-level can not work with blocking locks.

There are two parts to the solution. As in the original code, when the packet arrives, the interrupt handler determines the packet types and selects a destination queue for the packet (*e.g.*, *ipintrq*). These queues are instances of *ifq*s, manipulated by a well-defined set of macros. We modified those macros (*IF_ENQUEUE()*, *IF_DEQUEUE()*, *etc.*) to operate in a multiprocessor environment using spin locks so that the macros could be used without change at interrupt-level and in thread context.

Having queued the packet, we awaken a *netisr_thread*. The *netisr_thread* invokes the appropriate protocol's incoming packet processing routine (*e.g.*, **ipintr**) and normal packet processing continues except that the packet is now handled in thread context rather than at interrupt-level. Multiple *netisr_thread*s permit parallel processing of incoming packets; the number of *netisr_thread*s is configurable.

For historical reasons, a separate thread was invented to handle incoming ARP requests. This thread could be eliminated today but there is no strong reason to do so.

There were a number of other, lesser problems with interrupt handling that we do not have space to recount. The problems we have discussed have been the most interesting and the most representative.

3 Filesystem Parallelization

The 4.3BSD filesystem code distributed with Mach is essentially identical to the filesystem code distributed by Berkeley. Some small modifications have been made at CMU but the scope of those changes is small and therefore irrelevant to our discussion. The following discussion applies to generic 4.3BSD-based filesystems.

3.1 Design Rules

Wherever possible, we exploited "natural" data structure parallelism. It was clear that the filesystem offered significant opportunities for data structure parallelism: *a priori*, there was every reason to believe operations could proceed in parallel on separate disks, filesystems, file descriptors, file structures, inodes, buffers, *etc.* It was also clear that operations could proceed in parallel against separate elements within important tables, like the inode and buffer cache hash chains. Most importantly, the natural structuring of the filesystem code implied that there were few potential deadlock problems between locks held at the various filesystem layers. For example, a thread could acquire (in order) a file structure lock, an inode lock, a buffer lock and device driver locks without deadlocking with other threads performing similar activities. On the other hand, there were some interesting races within the various layers. There were small but easily resolved problems with interrupt-level code (see Section 2).

We did not need to re-design any of the existing 4.3BSD filesystem data structures, even where those data structures were internal and had no on-disk representation.

Initially we used only blocking, mutual exclusion locks to simplify implementation and ease debugging. As the code matured we migrated to read/write and *simple_lock*s.

In the Encore Mach/0.5 release, most filesystem code has been parallelized, including the tty subsystem and all interrupt-handling code. There are a number of subsystems that remain unparallelized. The various CMU-developed remote filesystems, RFS and VICE, have been modified to work in conjunction with the parallelized filesystem code, chiefly by taking and releasing filesystem locks at the appropriate times. This is not to say that these subsystems have been parallelized; they still depend on the *unix_master* restriction because the RFS- and VICE-specific code and data structures

have not themselves been parallelized. Other major subsystems that have not been treated include quotas and a CMU-specific pseudo-tty implementation.

3.2 Implementation Details

The scope of the filesystem parallelization effort is too broad to recount in detail. Instead, we will discuss some of the interesting cases encountered in the implementation. (Refer to Appendix II for a list of the various locks added to the 4.3BSD code in Mach/0.5.)

The most challenging subsystem to parallelize turned out to be the buffer cache. The relationships among the hash table, the various freelists, and the buffers themselves are complex and further complicated by the different ways the cache can be accessed from interrupt-level and from within thread context. Interrupt-level buffer cache manipulations had to be eliminated, as we described in Section 2.2.

The internal complexity of the buffer cache led to a large number of possible deadlocks. Most of these deadlocks were resolved without restructuring the underlying algorithms by using conditional locking. With conditional locking, a thread receives an error indication if acquiring a lock would require blocking. For example, when fetching a disk block from the cache, it is necessary to lock the hash chain where the buffer containing the block should go, search the chain and, on a miss, allocate an empty buffer from the free list. However, buffers on the free list are also linked onto hash chains and must be removed from those chains. Naively acquiring the second hash chain lock could deadlock. Releasing the first hash chain lock opens up new races and at a minimum requires re-locking and re-searching the hash chain after a buffer has been allocated from the free list. We chose to attempt a conditional lock on the second hash chain and, if the lock attempt failed, to try allocating a different buffer from the free list.

The buffer cache returns locked buffers to callers, so that the calling code doesn't have to be modified to understand buffer locking. A substantial amount of code did *not* have to be altered because of this implicit locking. For example, cylinder group information is fetched through the buffer cache and operated on within the buffer itself. The buffer lock implicitly protects the cylinder group data, permitting significantly easier parallelization of the disk block allocation and de-allocation code.

That very same disk block allocation code provides a good example of the use of our parallelization framework. At an early stage in the filesystem parallelization process, all of the disk block allocation code was single-threaded through a disk block allocation lock (*disk_alloc_lock*). This scheme allowed us to bring up the filesystem quickly as only the few routines used outside of the disk block allocation package (*e.g.*, **bmap**, **ialloc**, **ifree**, and **dirpref**) had to be modified to take the *disk_alloc_lock*. There were no race conditions to consider and the implementation took very little time. Once we had the filesystem running and had achieved basic stability we revisited the disk block allocation issues and migrated to a scheme using the implicit cylinder group locks described above. However, it was also necessary to lock accesses to the in-core superblock at appropriate times and guarantee that there were no deadlocks between superblock locks, (implicit) cylinder group locks and other filesystem locks.

At a higher level, we encountered a number of interesting problems with file descriptors and file structures. Mach permits all of the threads in a task to share the task's file descriptor table. It is then possible for one thread in a task to be altering the descriptor table while another thread is using it. We defined individual locks for each file descriptor to allow as much parallelism through this table as possible. (We envisioned utilities like parallel make, find, and grep that would be heavy file descriptor table users.) The individual locks created their own problems: for example, two threads within the same task trying to **dup2**(2) could deadlock trivially if the first thread attempted a **dup2**(2)(X,Y) while the second thread attempted a **dup2**(2)(Y,X). For any situation requiring the

acquisition of two file descriptor locks, we ordered the lock attempts by lock address to guarantee that no deadlock could result.

The interactions between pathname to inode translation (**namei**), inode fetching (**iget**) and filesystem attaching and detaching (**smount**, **umount**) become slightly more complex in a multiprocessor environment. **Iget** must cross mount points from the top of the filesystem hierarchy on down; **iget** detects mounted-on inodes and automatically fetches the root inode of the mounted filesystem. **Namei** performs the opposite task: when translating ".." in pathnames it occasionally must cross a mount-point going back up the filesystem tree.

In both cases, the original code "knew" that a filesystem could not be added to or removed from the mount table while **namei** or **iget** was active. In our multiprocessor kernel that assumption becomes invalid. The mount table was given a read/write lock, satisfying two constraints:

- provide maximum parallelism for frequent operations, *viz.*, **namei** and **iget**
- add minimal complexity to **smount** and **umount**

Had we used a mutual exclusion lock, **namei** and **iget** would have serialized across mount-points. On the other hand, a flag-based mechanism or some other lock that couldn't be held across an I/O would have significantly complicated the **smount** and **umount** code. By taking the *mount_table_lock* for writing, the **umount** code prevents **namei** and **iget** from crossing mount-points, thus making it easy to determine whether a filesystem is inactive. **Smount** holds the *mount_table_lock* write-locked to eliminate other races. Since **smount** and **umount** are both infrequent operations, the typical case where the *mount_table_lock* is held read-locked presents no bottleneck whatsoever.

There were a number of minor annoyances related to the use of global variables. One embarrassing instance occurred with the **bmap** subroutine. We overlooked the read-ahead variables, *rablock* and *rasize*, maintained so that the callers of **bmap** know what block to request on a read-ahead operation. This omission on our part turned out to be insidious: for a very long time we weren't aware that there was any problem at all. The read-ahead variables were frequently clobbered by another thread before they could be used by the thread that originally set their values. The resulting buffer read-ahead calls were nearly useless. Because the failure resulted in decreased performance but not in system failure (panic) we had no reason to suspect the existence of the problem. In fact, the problem was finally detected only because we noticed an unusual number of read-ahead calls into the buffer cache for disk blocks that had no business being the targets of read-ahead operations. We eliminated the global variables and forced **bmap** users to supply call-by-reference read-ahead variables.

Encore Mach/0.5 eliminated the *unix_master* restriction for roughly four dozen frequently used filesystem calls (some of which are simply separate entry points to common subroutines):

- open, creat, close, read, readv, write, writev, lseek
- link, unlink, symlink, readlink, mknod, mkdir, rmdir
- chdir, chmod, fchmod, chown, fchown, chroot, saccess
- stat, lstat, fstat, dup, dup2, ioctl, fcntl
- rename, truncate, ftruncate, utimes, flock
- smount, umount, sync, fsync, execv, execve
- swapon (a no-op under Mach)
- pipe (also required network parallelization)
- select (also required network parallelization)

3.3 Performance Analysis

3.3.1 The Benchmark

The performance analysis effort used the Neal Nelson Business Benchmark[4], a commercially-available set of system benchmarks. The NNB is oriented towards traditional Unix filesystem operations. While Mach has a notion of memory-mapped files (and this notion has become popular in various Unix dialects) we were more interested in characterizing the improvements we had made to the 4.3BSD compatibility code. The NNB fit the bill: it is simple to use, popular, and results are available for a wide variety of systems. (Note that the results we obtained are used only for comparisons internal to Encore, and that the data derived from the NNB suite is reprinted here with the permission of Neal Nelson and Associates.)

The Neal Nelson Benchmarks consist of 18 separate tests oriented towards measuring filesystem and processor performance. Space limitations force us to confine our discussion to only four of those tests. Here are brief descriptions of them:

Test #1. "The Average User": various calculations and filesystem functions intended to represent the average user at work.

Test #3. Disk I/O: 250 iterations of a loop with a mixture of filesystem I/O functions.

Test #8. 500K Function Overhead Loop: call an empty function many times.

Test #18. Random Disk Tests: random reads from the disk.

The NNB driver is compiled with an option to select the maximum number of users to simulate during the benchmark run, typically between 20 and 60. During the course of the run, the driver executes a test program with arguments that select one of the 18 tests. The driver begins by executing one copy of the test program and recording the completion time for the test. The driver then executes two copies of the test program, as nearly simultaneously as it can manage, and records the completion times for those tests. This process is repeated until the driver has executed up to the maximum number of test copies requested.

3.3.2 Test Conditions

The NNB suite was run on a Multimax-320 configured as follows:

- 3 APC-01 CPU boards, 2 two-MIPS NS32332 CPUs per card, total 12 MIPS
- 2 SMC-16 memory cards, at 16 megabytes each, total 32 megabytes
- 1 EMC-I, with one Ethernet interface and one masstore interface
- 1 NCR disk controller
- 1 CDC Sabre 1.2 gigabyte disk drive, with average access time of 8.3 ms.
- 1 SCC, the System Control Card (irrelevant to this discussion)

As with all NNB runs, the system was brought to multi-user mode and a representative of Neal Nelson Associates downloaded and executed the benchmark. There were no other users logged in. There was substantial overall network traffic but only broadcast packets were sent to the benchmark machine. Network packets were therefore processed by the system; however, we presume that all benchmark runs should have been affected to approximately the same extent. We also ran unofficial

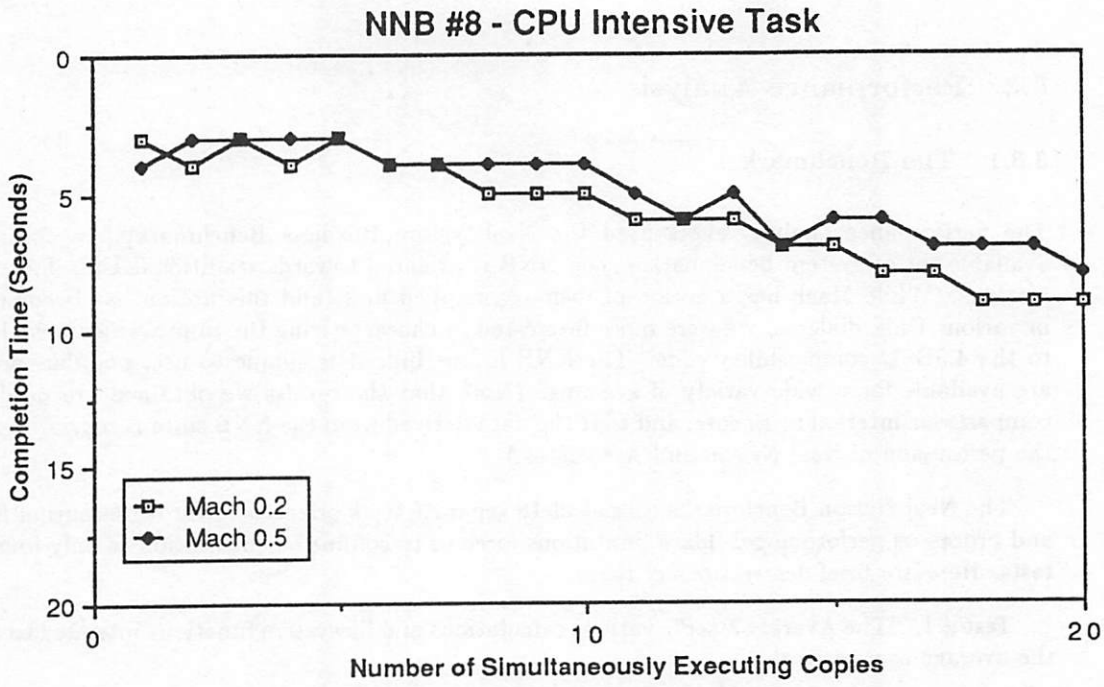


Figure 1: CPU-Bound Jobs under Mach/0.2 and Mach/0.5

benchmarks from single-user mode with the network interface disabled and achieved nearly-identical results; the differences were statistically insignificant. A single *biodone_thread* was present and active as needed. The *slcintr_thread* was present and would have been active whenever the console presented input to the system so the console was not used.

Both Mach/0.2 and Mach/0.5 booted from the same root partition and shared the same user partition. The NNB suite resided on the user partition and all working files for the suite were contained on that partition, as well.

The NNB was compiled for 20 users. (At larger numbers of users, the tests take a very long time to run. In the future, we hope to have the opportunity to reserve a test machine for sufficient time to run a 60 user test.) The entire suite was run against Mach/0.2, the "serial" kernel, and Mach/0.5, the "parallel" kernel.

3.3.3 Test Results

The overall results indicate that Mach/0.5 does a substantially better job of exploiting the parallel architecture of the Multimax than does Mach/0.2. We will discuss some specific cases first and close with the most general test.

The compute-bound tests, such as NNB #8 (see Figure 1), revealed no significant performance improvement in Mach/0.5 over Mach/0.2. Although the graph shows a small difference between Mach/0.5 and Mach/0.2, the difference is largely attributable to round-off error. All of the tests are coded to record only the time consumed by their CPU-bound portions, and both Mach/0.2 and Mach/0.5 distribute user-level computation to any available processor, so both versions of the operating system delivered similar results on the compute-bound benchmarks.

NNB #18 yields more interesting results (see Figure 2). This test lseeks and reads from different parts of a working file. (Each simultaneously executing copy of the test has its own working file.) The test demonstrates a significant performance improvement for roughly 6-10 simultaneously executing copies of the test. However, Mach/0.2 degrades more slowly than we would expect and at roughly eight simultaneous tasks Mach/0.5 degrades surprisingly quickly, approximating the performance of

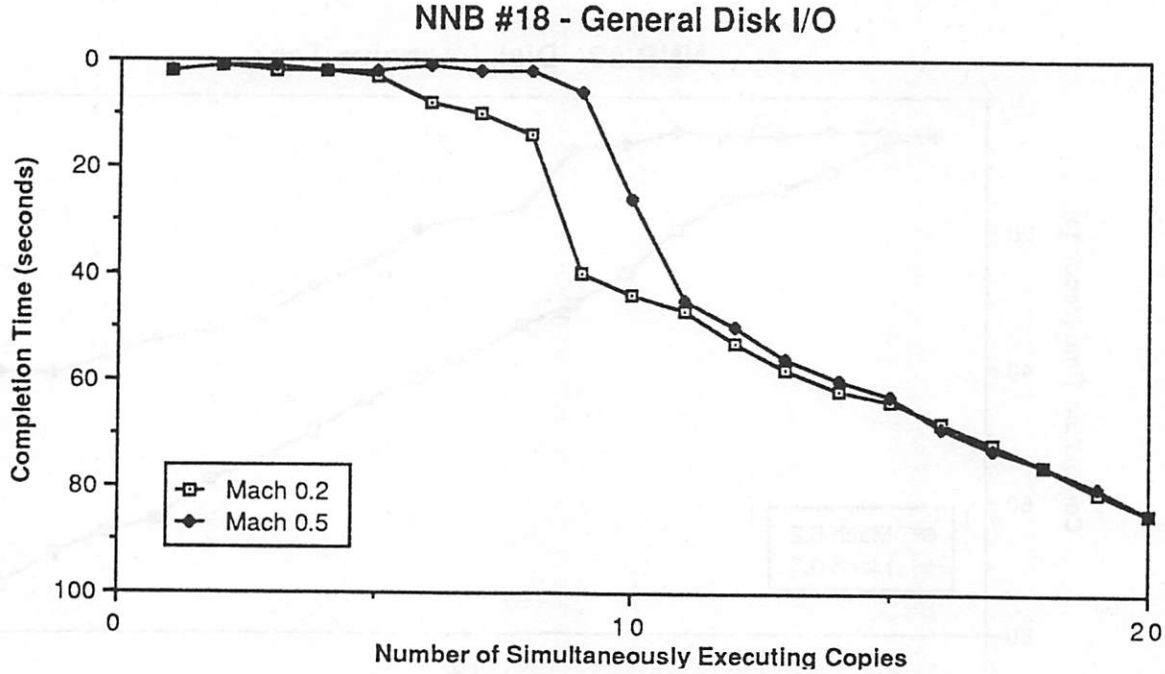


Figure 2: Random Disk Tests under Mach/0.2 and Mach/0.5

Mach/0.2 from eleven through twenty simultaneous tasks. This benchmark may indicate that the directory's *inode_lock* becomes a bottleneck. Alternately, we may be seeing saturation of the disk channel. A third possibility is CPU saturation. The most likely culprit, though, is the *bfreelist_lock*, which statistics demonstrated had a miss ratio an order of magnitude worse than the next most frequently used lock.

NNB #3 tests disk I/O by explicitly seeking to the beginning of the working file and performing five sequential 512-byte reads followed by five sequential 512-byte writes, after which random seeks and reads are done against the working file. This loop is repeated 250 times. Once again, each task has its own working file. Mach/0.5 clearly out-performs Mach/0.2 until about eight simultaneous tasks, when decay sets in (see Figure 3). While *inode_lock* contention or disk channel saturation may be possible, as with test #18 the most likely culprit seems to be the *bfreelist_lock*, which once again demonstrated an unusually high miss ratio.

NNB #1, representing the average user at work, nicely summarizes the current level of filesystem parallelization (see Figure 4). While the Neal Nelson Benchmark suite suggests that Mach/0.5 suffers from one or more as-yet-unidentified hotspots, Mach/0.5 represents a substantial improvement in filesystem parallelism over Mach/0.2. We have already benefited from our incremental approach to parallelization by quickly bringing up a working system and then concentrating on parallelizing the worst bottlenecks first.

3.3.4 Future Work

Future filesystem parallelization enhancements will be guided chiefly by analysis of lock contention statistics to detect bottlenecks. Undoubtedly some of this work will focus on reducing *bfreelist_lock* contention as well as on improved inode and buffer locking. Selective use of inode read locks could dramatically increase parallelism on commonly-used files and directories and could be achieved with small modifications to **namei**, **iget**, and **rwip**. An additional interface to the buffer cache could be provided for the case where a buffer is going to be read but not written. (**bread** must assume that the buffer will be modified by the caller.) In this case, the buffer cache could read-lock the buffer, allowing it to be shared by other readers.

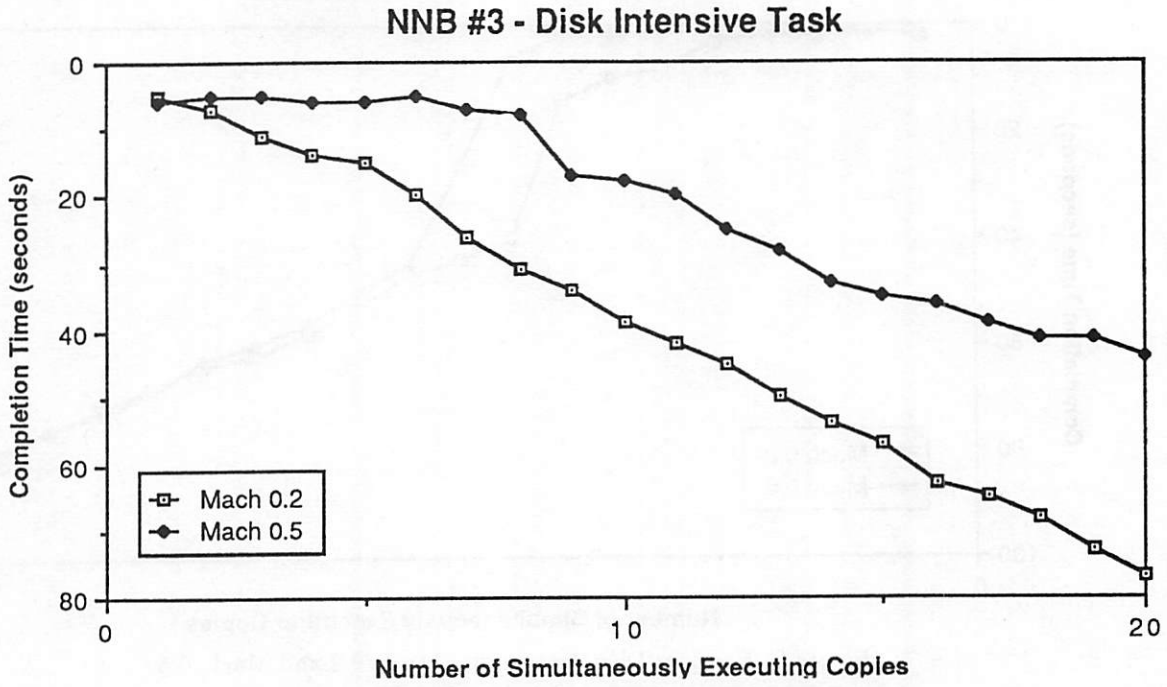


Figure 3: More Disk I/O on Mach/0.2 and Mach/0.5

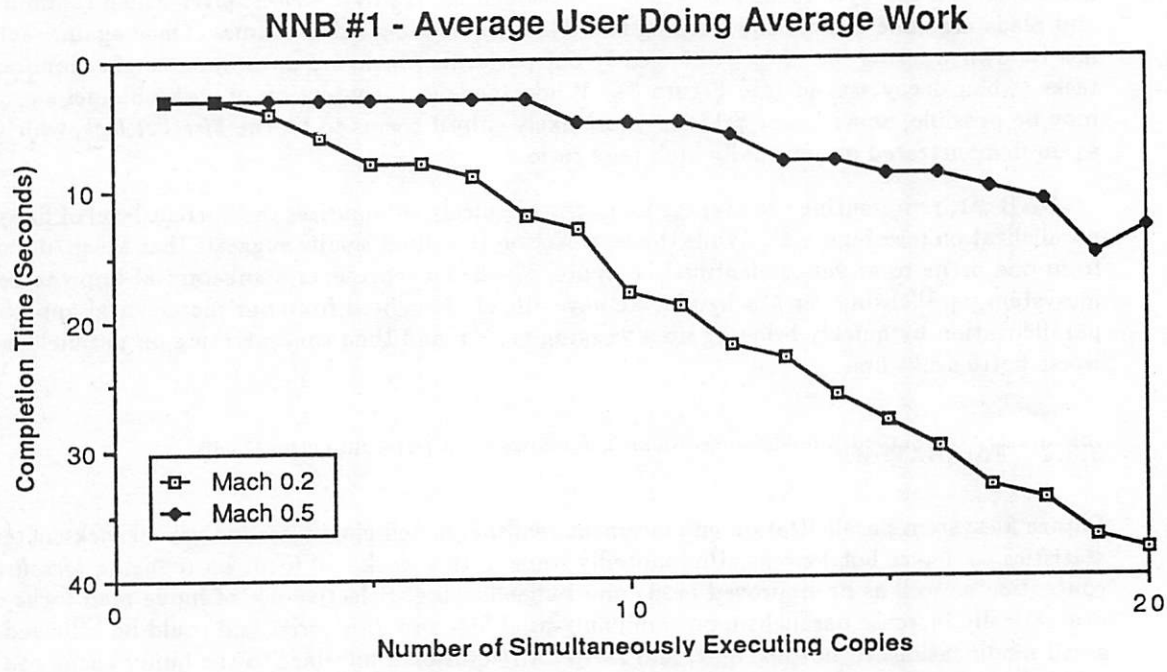


Figure 4: The Average User Working under Mach/0.2 and Mach/0.5

More aggressive optimizations are conceivable. For example, inode locking as a means of preventing simultaneous overlapping modifications of file data largely could be eliminated. Buffer locking can synchronize modifications to the same block of file data. Inode locking could be restricted to the cases where the file's size would change or the I/O would span multiple file blocks. An optimization of this nature might have a beneficial effect on database operations against large, random-access files.

Finally, the direction of our work will change somewhat as we incorporate the latest CMU release of Mach, which contains a vnode layer and client and server NFS. This work is already well under way and has had a major impact on filesystem locking strategies.

4 Network Parallelization

Parallelization of the network subsystem was accomplished by dividing the network code into the same layers as defined by the ISO/OSI 7-layer model. Each layer, Link (device driver), Network (IP, ARP), and Transport/Session (TCP, UDP) was examined and parallelized separately. By so doing, we realized two benefits. First, multiple developers could work on separate sections of code with only minimal interference. Second, lock contention and overall performance could be examined and effort applied to only those algorithms or data structures revealed to be bottlenecks.

4.1 General Lock Policy

The network code presented a fundamental problem for parallelization: not only could data transfer be initiated by the local user but also asynchronously from the network. In other words, the user may send packets to the network interface whenever he wishes and (from the standpoint of the kernel) the network interface may send packets whenever it wishes. This behavior is different than that of the filesystem where interrupts do not generally represent unsolicited I/O operations but the completion of a user-initiated event.

Rather than poll the network interface for new packets, the 4.3BSD code, triggered by a network interrupt, pushes the packet across multiple protocol layers all the way up to the socket queue. In a kernel using locks to serialize destructive transactions, care must be taken to prevent the obvious deadlocks that can result from threads simultaneously traversing these layers in opposite directions.

To prevent deadlocks, permit multiprocessor execution, and encourage a speedy initial implementation, we decided upon a straightforward locking policy: each protocol would have a single, global lock guarding its data. A protocol's lock would be taken when using any associated protocol code and released when the protocol invoked a lower or higher layer. A thread that could not immediately acquire one of these locks would be put to sleep and is woken when the lock became available. This scheme was sufficient for protocols such as ARP which have little traffic, but not acceptable for IP, TCP and UDP where there is significantly more traffic. For these "high-use" protocols, we ultimately developed finer-grained locking schemes on a per-connection basis.

The protocols we parallelized included TCP, UDP, ICMP, ARP and IP. We did not have the time or the need to also parallelize other protocols present in the 4.3BSD distribution, such as Xerox NS or VMTP from Stanford.

A number of asynchronous kernel threads were created to handle timer based events for the various protocols. Under 4.3BSD all timer based operations, such as connection time-out, keep-alive transmission, and packet retransmission are performed at interrupt-level from the callout queue. As these actions may need to take locks, all such operations were moved into separate kernel threads. The function of the callout queue entry is now, simply, to wake up the appropriate kernel thread.

4.2 Link Layer

The link layer primarily consists of device drivers. The Multimax uses intelligent controllers for all I/O operations, including Ethernet. Refer to Section 2.3 for the details of interaction with the Ethernet device driver.

4.3 Network layer

The network layer consists of the IP, ARP and ICMP protocols.

4.3.1 ARP

ARP packets are handled by two kernel threads with a single global lock around all ARP data structures. One of these threads processes incoming ARP packets; the second thread is used to time out old entries in the ARP table. While finer-grained locking has been considered, analysis of lock statistics shows that there is little lock contention in this area and we have concentrated our efforts elsewhere.

4.3.2 IP

The IP code is almost completely free of locks. Most packets pass through the IP layer without ever taking a lock. The major exception is packet fragmentation and reassembly, which is controlled by a single lock. On networks where there is a great deal of IP fragmentation, this single lock may be a bottleneck; however, on most local area networks there is no IP fragmentation. Even our Internet connection receives only an occasional IP fragment.

A separate kernel thread was created to handle IP timeouts. The only use of these timeouts is to remove old fragments from the queue. A thread was required as the IP lock needs to be held during this operation.

One interesting problem existed with incoming source routes. These are IP options to be used in replies to the incoming message. The original 4.3BSD implementation used a static structure to contain this information. As IP is a state-less protocol, there is no "connection" information maintained. A classic uniprocessor assumption was made that no other thread could change the data before the reply was sent.

With no per-connection structure to store this information, a place needed to be found to store the information. The solution used was to save the information in Mach's equivalent to the 4.3BSD *u-area*.

4.3.3 ICMP

The ICMP code is similar to IP in that few locks are required. In fact, the only lock is in the case of REDIRECT requests, *i.e.*, changes to the route table. The route table is protected by a read/write lock.

4.4 Transport/Session layer

The TCP and UDP protocols were parallelized in almost identical ways. For both of these protocols a linked list of all connections is maintained. In the Mach/0.5 implementation described in this paper, a mutual exclusion lock protects all operations to this list, including lookups. A new version of the kernel which uses read/write locks has already been implemented to allow simultaneous lookups.

Once the connection is found (with **in_pcblookup**), a reference count in the per-connection *inpcb* structure is incremented (preventing the deallocation of the structure), the global lock is released and the *inpcb* lock acquired, thereby guarding the connection against simultaneous access. This lock is held during all packet processing. While it may be possible to release the lock, or to use a read/write lock, current statistics do not suggest that such a change is warranted.

The single major difference between TCP and UDP is that TCP provides reliable data transfer. This implies the need for retransmission, maintaining connections, *etc.* Much of this activity is driven from two timers; "fast" (200ms) and "slow" (500ms). As the TCP connection chain must be traversed during these timeouts and locks taken, separate kernel threads were created to handle each of these timeouts.

4.5 Miscellaneous

The user layer and protocol layer are quite separate in the 4.3BSD model. The user layer interacts through system calls such as **read(2)**, **write(2)**, **send(2)**, and **recv(2)**. Each of these calls ultimately uses a *socket* structure, each of which now has its own lock. All operations on the socket are protected by this lock. When the user sends data, the data is chained to the socket while the socket lock is held. Receive operations dequeue data from the socket, also under lock. Lower level protocols that work with sockets, such as TCP and UDP, must not only take the relevant *inpcb* lock but any appropriate socket locks as well.

The network memory pool is almost exclusively made up of *mbufs*, which come from two pools, the mbuf list and the cluster list. Mbufs may be allocated or deallocated in both interrupt and thread context and so each list has its own simple lock. Although mbufs are used widely in the 4.3BSD code, the implementation simply required the addition of locking calls to a few macros and supporting subroutines.

Under 4.3BSD UNIX pipes use sockets for I/O. As a direct result of the network parallelization, pipes also operate in parallel.

4.6 Parallelized Network Calls

The network parallelization effort allowed a large number of 4.3BSD calls to execute in parallel and permitted outgoing and incoming packets to be handled on any processor. The affected system calls included:

- accept, bind, connect, listen, shutdown
- recv, recvmsg, recvfrom, send, sendmsg, sendto
- socket, socketpair, pipe
- getsockopt, setsockopt, getpeername, getsockname

4.7 Network Performance Analysis

There are many components within the network subsystem that affect performance. While we would have liked to measure the performance of individual pieces of the network code, for our purposes here we present an analysis based on total TCP throughput. Unfortunately, there are no standard network performance tests similar to the disk I/O tests performed by the Neal Nelson Benchmarks. Therefore, we constructed our own network performance tests.

The fundamental test we developed creates a TCP connection to a remote system and repeatedly sends data using the write(2) system call. The recipient simply reads and discards the data. The size of the write requests was varied using values of 1, 2, 10, 64, 100, 512, 1000, 2000, and 16K bytes. During the development of these tests we experimented with other values but did not find that they yielded much additional information. The total amount of data sent was controlled so that the length of the test was at least five seconds and ran no more than ten minutes. These times were chosen to provide steady-state performance without forcing the benchmarking process to become needlessly lengthy. Only time to transfer the data was counted; time to establish and close the connection was not included. For each request size the experiment was repeated three times and the average of the three runs was used in the accompanying graphs.

The test just described uses only a single TCP connection. We created another test using multiple copies of the single-stream test. Data was also collected while running 2, 3, 5 and 10 simultaneous copies. As before, the multiple connection experiments were run three times and the average of the three runs was used.

The systems used to run these tests were two Multimax-320 systems, each configured as follows:

- 4 APC-01 CPU boards, 2 two-MIPS NS32332 CPUs per card, total 16 MIPS
- 5 SMC-16 memory cards, at 16 megabytes of memory, total 80 megabytes
- 1 EMC-I, with one Ethernet interface and one masstore interface
- 1 CDC Sabre disk drive
- Private Ethernet connection between these two machines

Baseline measurements were taken using the Mach/0.2 "serial" kernel (See Figure 5). For each request size from one through 512 bytes there was almost no increase in aggregate throughput when the number of connections was increased. Aggregate throughput only increased with additional connections when the request size exceeded 1000 bytes, and then by only 17% (1000 byte requests) to 42.5% (16K byte requests). As expected, the *master* CPU, forced to process all interrupts and incoming packets, as well as TCP, IP, and ARP requests was limited in the amount of network traffic it could handle. The performance improvement observed with larger packets resulted from the amortization of the (fixed-size) TCP/IP packet overhead across a larger quantity of data.

Analysis of the Mach/0.5 aggregate throughput (see Figure 6) shows that increasing the number of connections increases the aggregate throughput. For example, when making 1000 byte requests (typical for FTP) two simultaneous connections had 83% additional throughput over a single stream; obviously, the theoretical maximum would be 100%. Ten simultaneous connections had 517% additional throughput.

Many multi-processor benchmarks attempt to attain linear speedup as the number of simultaneous tasks increase. While that is also true with the network subsystem, the network has additional constraints that a CPU-bound benchmark does not; most importantly, the speed of the transmission line. Unbounded linear speedup, in this case, is not possible. Our tests were run using standard 10M bit/second Ethernet. The maximum theoretical data throughput of 1.25M bytes/second does not

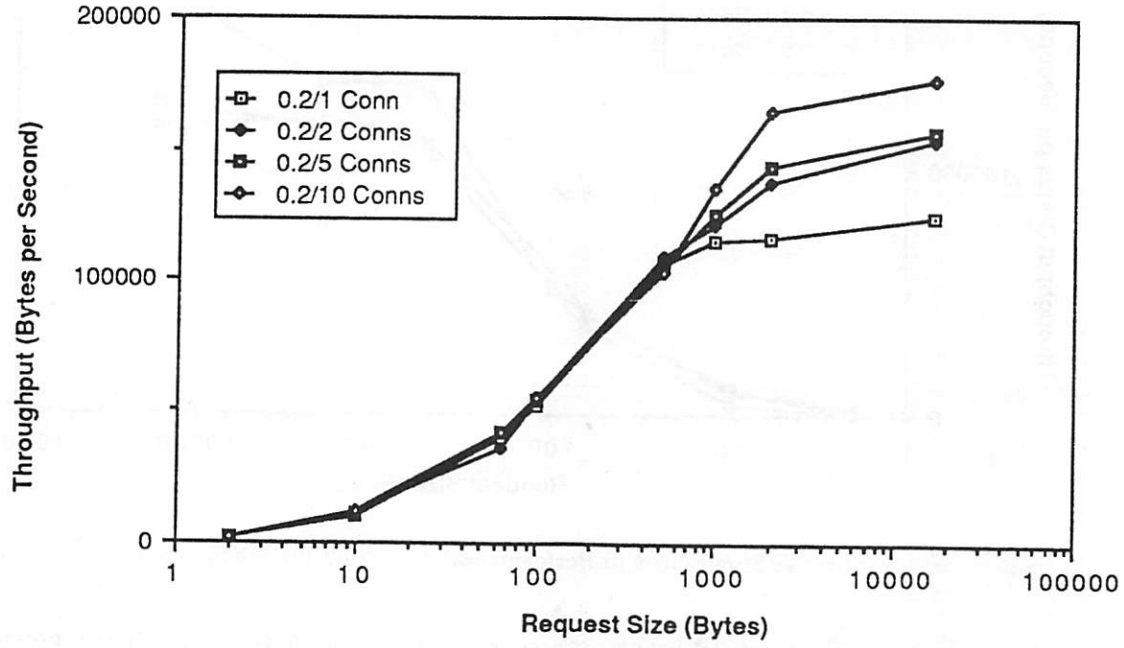


Figure 5: Mach/0.2 Network Performance

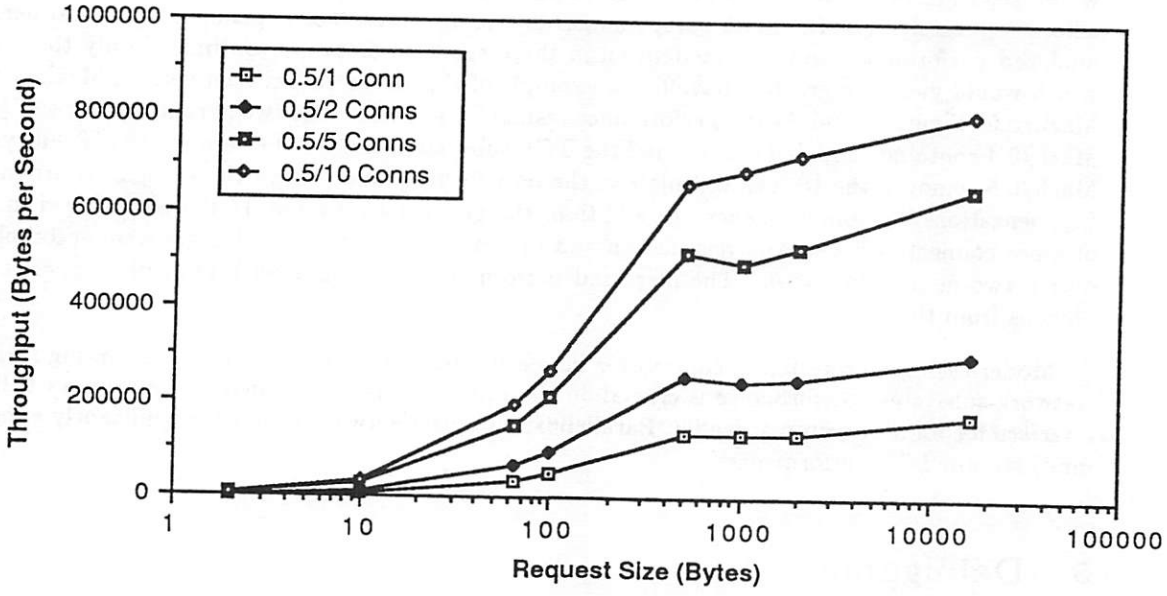


Figure 6: Mach/0.5 Network Performance

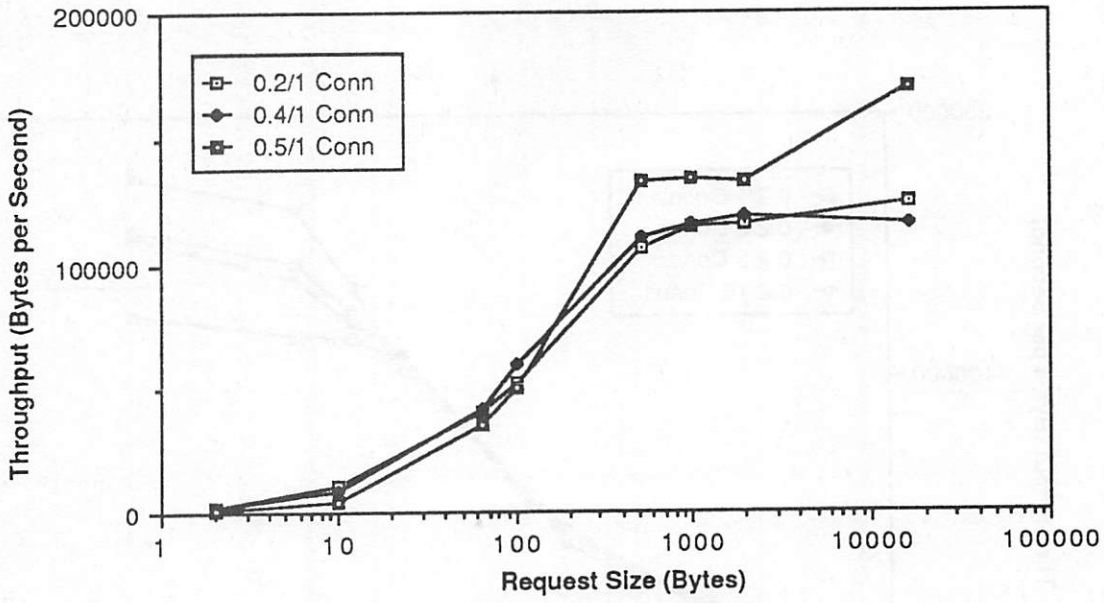


Figure 7: Single-Stream Performance, Mach/0.2 *vs.* Mach/0.5

take into account TCP header, IP header, source and destination address, CRC bytes, pre-amble, and collisions. In addition, the TCP protocol also requires acknowledgments from the receiver, each of these requiring a 64 byte packet. Given all of this, the effective maximum transfer rate is much closer to 1 Million bytes per second. The tests described in this paper show a maximum throughput of approximately 803,000 bytes per second, with every sign that additional connections could be supported, further increasing throughput.

As we have mentioned, the design of the network parallelization was done under a framework where separate functional areas of the network, such as IP, ARP, TCP and UDP were all parallelized separately. For the most part, changes in one area were not dependent upon another. We analyzed performance and lock contention in these separate areas and optimized only those areas which would yield the greatest payoff. An example of this occurred between version Mach/0.4 and Mach/0.5. Figures 7 and 8 show performance results for the serial and two parallel versions of Mach. Mach/0.4 contained a global lock around the TCP subsystem and another around the IP subsystem. Mach/0.5 removed the IP lock completely; the only locking done within the IP layer is around the fragmentation/reassembly queues. In addition, the global lock around TCP was removed in favor of a per-connection lock. Analysis, design and implementation of these changes were accomplished over a two-month time span. The increased performance, especially with multiple connections, is obvious from the graphs.

Modern computer systems require ever increasing performance from their networking facilities. Network subsystem performance is crucial on the Encore Multimax, which depends on an Ethernet interface for all user terminal traffic. Parallelization of the network code has significantly enhanced multi-stream TCP performance.

5 Debugging

Encore has created a number of tools to assist in the debugging of multiprocessor kernels. First, our standard user-level, high-level language debugger has been modified slightly to understand remote kernel debugging. All Encore operating system kernels include a very low-level, nearly stand-alone

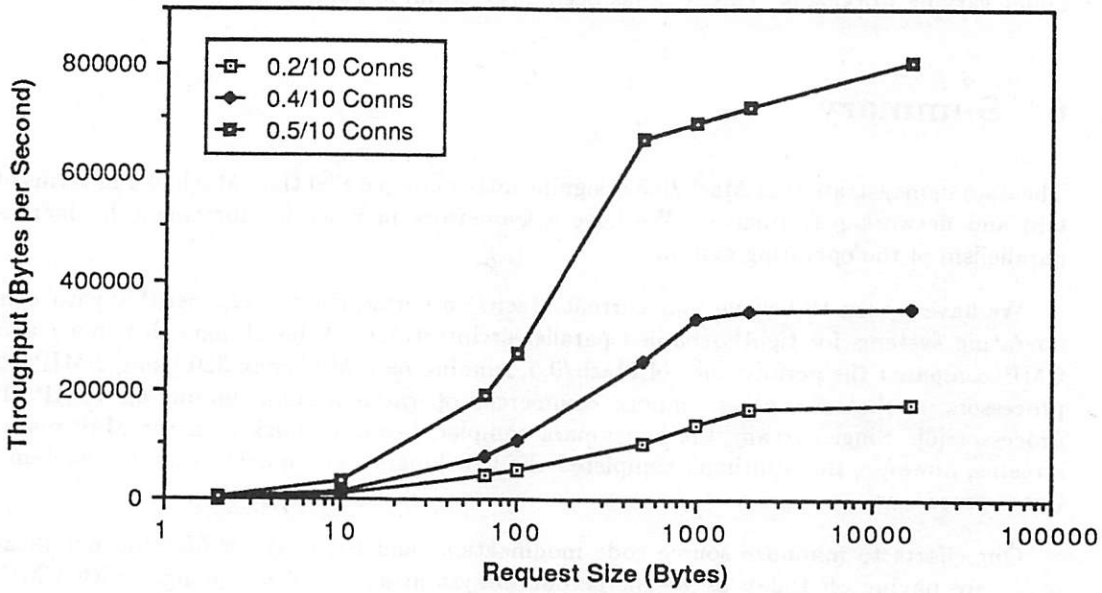


Figure 8: Aggregate Performance Gained by Incremental Parallelization

debugging module that understands how to observe and control the execution of the larger kernel. This debugging module communicates over a serial line with a production machine running our high-level debugger. The module permits single-stepping, tracing and observation of the activities of any processor on the machine being debugged. The high-level debugger allows the user to control the target kernel at the level of C statements or assembly-language instructions. In fact, the very same debugging module and high-level debugger are used to debug our low-level firmware and diagnostic code. Needless to say, these tools are invaluable.

For our project, we also developed a standard approach to coding locks. All locks are coded as macros, so the developer may modify a single definition to include extra debugging code or even, on occasion, to change the type of lock being used. A single, compile-time option indicates whether extra lock debugging code is to be included in the kernel image. Another compile-time option causes the locking routines to record statistics about lock contention rates.

When compiled for lock debugging, the lock routines themselves record the program counter where the lock was locked and unlocked but only for mutual exclusion locks, which is why many of our locks start out as mutual exclusion locks and are changed to read/write locks after being debugged. The lock routines also record lock ownership and check whether locks are being re-taken by the same owner or being released without having first been acquired (two common errors). Note that the locking routines will always record lock ownership, regardless of compile-time options. Lock ownership is a valuable clue when analyzing crash dumps.

Frequently, a function will include at its beginning debugging assertions about the state of various relevant locks. Especially important are assertions about locks that are expected to have already been taken by another routine. Such assertions prevent the vexing problem of unruly threads clobbering unlocked data. If any of these assertions fail, the kernel panics.

The blocking lock routines optionally track interesting lock statistics, including number of attempts, misses, forced re-schedules, minimum and maximum wait times, and total time threads spent waiting. We will soon have similar statistics on *simple_lock*s.

These statistics can be retrieved and displayed at any time with a simple user-level utility, allowing us to dynamically monitor a running system to detect locks with high contention rates

under varying workloads. This tool has been quite useful in guiding our parallelization efforts.

6 Summary

The data demonstrate that Mach/0.5 is significantly more parallel than Mach/0.2 in terms of filesystem and network performance. We have a framework in place for incrementally increasing the parallelism of the operating system.

We have reason to believe that current Mach/0.5 performance is competitive with commercial operating systems for tightly-coupled parallel architectures. A benchmark developed and run at CMU compared the performance of Mach/0.5, running on a Multimax-320 using 2-MIPS NS32332 processors, to that of another vendor's commercial operating system running on 4-MIPS Intel 386 processors[6]. Single-stream, the benchmark completed half as quickly on the Multimax. By ten streams, however, the Multimax completed the benchmark more quickly than the system built on faster processors.

Our efforts to minimize source code modifications and to always *#ifdef* the modifications we made are paying off today as we merge our filesystem and network changes with CMU's latest enhancements, including new networking features and a vnode layer for the filesystem.

Future work will focus on further improving the parallelization of Mach/0.5's 4.3BSD compatibility code. In particular, remaining frequently used or long-running system calls will be targeted for parallelization. Signal-related system calls are now at the top of our list. There are a number of other calls that only require *unix_master* because they depend on updating one or two 4.3BSD data structures (*e.g.*, the *proc* table) that are maintained chiefly for the benefit of user-level utilities that read kernel memory. In particular, **fork(2)** and **exit(2)** fall into this category.

Mach/0.5 is now in beta-test and soon will be distributed to the twenty-five Encore customers already running an earlier version of the parallelized filesystem and network code.

Appendix I: New Threads

Thread Name	Use
arpinput_thread	process incoming ARP packets
arptimeout_thread	ARP timeouts
biodone_thread	asynchronous filesystem I/O completions
ip_slowthread	time out old fragments
more_clusters_please	allocate more memory from virtual map for clusters
more_mbufs_please	allocate more memory from virtual map for mbufs
netisr_thread	thread context for handling incoming packets
rfsAbortOut_thread	handle CMU RFS aborts
rfsSendOut_thread	handle CMU RFS Send aborts
slcintr_thread	Multimax console serial line handler
tcp_fastthread	process delayed acks every 200ms
tcp_slowthread	update active timers, actions on timer expiration

Appendix II: New Locks

Lock Name	Type	Scope
accounting_lock	mutex	accounting functions
global_arp_lock	mutex	ARP data structures
bdl_count_lock	mutex	low-level disk I/O routines
bfreelist_lock	mutex	buffer free lists
bhash_lock	mutex	buffer cache hash chain
buf_lock	mutex	buffer
cfreelist_lock	mutex	cblock free list
console_lock	simple	low-level printing routines
console_printf_lock	simple	low-level printing routines
global_ip_lock	mutex	protocol
global_route_lock	read/write	Protects network route table
global_tcp_lock	mutex	protocol
global_udp_lock	mutex	protocol
global_unpconn_lock	mutex	make and break unp connections
fdesc_lock	mutex	individual file descriptor
filesys_lock	mutex	filesystem structure
file_table_lock	mutex	global file table
fstruct_lock	mutex	single file structure
hostname_lock	read/write	hostname variable
ifreelist_lock	mutex	inode free list
inode_hash_lock	mutex	inode hash chain
init_bdl_lock	simple	low-level disk I/O routines
inode_lock	mutex	inode
log_open_lock	mutex	error logging subsystem
mbuflist_lock	simple	mbuf allocation
mcllist_lock	simple	mbuf cluster allocation
mount_table_lock	read/write	mount table
nextinodeid_lock	simple	namei cache
panic_lock	simple	serialize panicky code
select_element_count_lock	mutex	select subroutines
socket_lock	mutex	individual socket
time_lock	simple	time variable
tty_lock	mutex	tty/pty structure
unp_gc_lock	mutex	unp garbage collection
uu_ip_lock	simple	IP source routing

References

[1] M. Bach and S. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63:1733–1749, October 1984.

[2] J. Barton and J. Wagner. Beyond Threads: Resource Sharing in UNIX. In *Winter 1988 Usenix Conference Proceedings*, February 1988.

[3] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel. In *Proceedings of the Winter USENIX Conference*, 1988.

[4] Neal Nelson and Associates. Neal Nelson Benchmark Report, 1986. Benchmark results reprinted by permission.

[5] R. Rashid. Threads of a New System. *Unix Review*, August 1986.

[6] R. F. Rashid. A Proposal to UNIX International to Integrate Mach Technology into UNIX System V, May 1989. Submission to UNIX International Multiprocessor Working Group.

[7] U. Sinkewicz. A Strategy for SMP ULTRIX. In *Summer 1988 Usenix Conference Proceedings*, February 1988.

Efficient Implementation of Modularity in RAID*

Charles Koelbel
Fady Lamaa
Bharat Bhargava

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Raid is a distributed database system that is very modular. This paper describes our design, implementation, measurements, and experiences in modifying the system to achieve an efficient implementation without sacrificing the original goals of modularity in the Raid system. This paper describes the rationale behind the modifications that lead to a facility by which several servers can be merged to run in a single operating system process. It includes an account of the changes that were needed in the different layers of the system. We feel that merging servers is a technique that can be applied to improve performance on any server-based system. However the modifications and alternatives for implementation are not easy to evaluate. This research study contributes in this direction. We have presented the data that was collected for database transaction processing using the old version (one server per process) vs. new version (multiple servers per process) in the Raid system.

1 Introduction

RAID is a robust and adaptable distributed database system for transaction processing [3]. The Raid system is based on a server-processing model similar to the CAMELOT [10], SDD-1 [9], and R* [6] systems. This model divides the functions of transaction processing into software modules called servers. A high-level communication package provides a clean, location independent interface between

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, by UNISYS and AT&T Corporations.

servers. Naive implementations of server based designs, however, often have high overheads for communications between servers. In RAID, for example, a single message round-trip typically takes 10 milliseconds; for comparison, a server's computation for a transaction may only take 40 milliseconds. System throughput could be improved if the interprocess communications time were reduced to a few milliseconds or hundreds of microseconds. This paper describes an approach to reducing this overhead which can be applied to any server-based system.

Raid's server design has facilitated the implementation effort by providing for flexibility, and by explicitly defining the interfaces between servers. This architecture provides for modularity and extensibility which in turn gives the capability to build an adaptable and dynamically reconfigurable system. The architecture has been criticized, however, as inefficient for production systems. We believe that this is not the case; rather, we feel that the server model is a valuable tool for creating modularity in the design of the system. Current software engineering practice strongly supports such methodologies. The work we report here shows that such modularity need not extract an unreasonable price in terms of performance. Thus, it points the way for greater use of the server paradigm in designing practical, modular systems.

The server model was used to implement two versions of the Raid system. The first version runs with each server in an asynchronous process. The second version combines the servers that do not need to be asynchronous into a single process. Since our objective has been to conduct scientific experiments and measurements on various protocols for transaction processing and system configurations the first version has been very convenient. However, its performance was not satisfactory, particularly in terms of the contribution of communications overhead. We decided to tune this system and build the second configuration with merged servers to see what improvements could be achieved. We wanted to experiment with various alternatives for the implementation of the merged server version and to gain useful experience for other research projects.

The original implementation of RAID had an attractive server model, but an inefficient implementation of that model. Our goals for the modification for the new version were to

- Decrease transaction latency in RAID by decreasing server overheads.
- Retain the same basic design.
- Retain as much modularity as possible.

We achieved these goals by merging conceptually separate servers into a single physical entity.

The original implementation of servers in RAID used one process for each server and a general-purpose communications protocol for interprocess communication. In addition to the communications overheads, this lead to excessive context switching, particularly since there are many servers in the system. There are several ways of reducing these overheads in RAID:

- *Use a special-purpose communications protocol between servers.* The protocol can be tailored to the needs of the current system, and need not pay a price for unused functionality. This reduces communications overhead, but does little for context switching. [1] discusses work on this idea done in RAID.

- *Write the system in an object-oriented language*, using the basic server design. This approach shifts the burden of message passing onto the language run-time environment, which is presumably more efficient than general-purpose operating systems. This is particularly attractive if the implementation language uses the same syntax for both local and remote object references, as Objective-C does [5]. Depending on the language, this may reduce communications overhead, context-switching overhead, or both. Emerald [4] is an example of a system that uses this method.

- *Implement servers as lightweight processes (threads)*, rather than full UNIX processes. The threads' shared memory space can then be used for message passing, rather than costly operating system primitives. This saves both communications and context-switching overhead. The Argus project [7] took this approach by building their own threads package. Many modern operating systems such as Mach [8] also have thread primitives.

- *Merge several servers into the same process* rather than using a separate process for each server. This approach attempts to convert inter-process communication into simple data movement within the same process. Communications overhead is reduced by this conversion, while context switching is reduced because there are fewer processes. The rest of this paper will focus on this possibility.

All of these methods are generally applicable to server-based systems, regardless of the communications system used (message-based, remote procedure call, etc.). The methods are not mutually exclusive; for example, it is reasonable to merge some servers and use a special-purpose protocol for communications between the other servers.

Performance gains for each of the above methods can be estimated. Using a special-purpose communications protocol may save 50% or more of the time spent on all communications. If 25% of the system execution time is spent in communications, this increases performance by 12.5%. Object-oriented languages have similar savings on communications, but tend to have higher overheads on other computations, so overall performance gains will tend to be lower. Good implementations of threads can reduce the time for a single context switch from hundreds of microseconds to tens of microseconds, and the time for a single communications call can fall from milliseconds to microseconds. If these two overheads together account for 50% of the system running time, performance gains can exceed 40%. The improvement for communications time using merged servers is similar to using threads for servers mapped into the same process; context switching between the merged servers is

eliminated entirely. The total time for these overheads will be less than the 50% we quoted for using threads, since not *all* communications or context switches are affected (unless all servers are merged). Assuming 40% of the total time is used by overheads between the merged servers, the performance gains would be about 30%. Similar analyses could be done for other server-based systems. In general, the higher the system overheads are, the more these techniques can gain in performance.

Note that merging servers does not change the basic server model; only the implementation of that model changes. (This is also true of the other changes noted above.) This is similar to the difference between the definition of the ISO network layers and non-layered implementations of the ISO standard [11]. Retaining the same server model is important, since experience with the original system can then be transferred directly to the new one. Also, it enables clear comparisons between the two systems. Merging servers will also create a system with good modularity if the merging is done intelligently; the conceptual servers become the modules of the implementation. This was very important in our work, since experimental systems like RAID must be adapted and extended frequently. Monolithic systems are notoriously difficult to modify. A final advantage of the merged-server approach is that it requires little or no modification to the server codes themselves. Merging servers does not change the servers' internal processing, only their packaging with respect to each other. Our implementation of RAID made no change to the server codes at all; only a high-level controller was added.

The next two sections describe the original and new RAID organizations in more detail. Section 4 then describes our implementation of the new organization. Section 5 compares the performance of the old and new systems. Finally, Section 6 gives some conclusions.

2 The Original RAID Implementation

The RAID system is organized in the hierarchy shown in Figure 1. In this paper we will focus on the server and communications layers. Each of the servers shown in the figure responds to service requests from the other servers in a clearly defined manner. The RAID communications package uses UDP to implement communications between servers. It provides a number of extensions to UDP, including

- A high-level naming scheme
- Location independence of servers
- Arbitrary sized messages
- Multicast support

The first two extensions are major advantages of the RAID communications package, and are closely related. Each server has a RAID address consisting of its type, site number, RAID number (several independent instances of RAID can be running

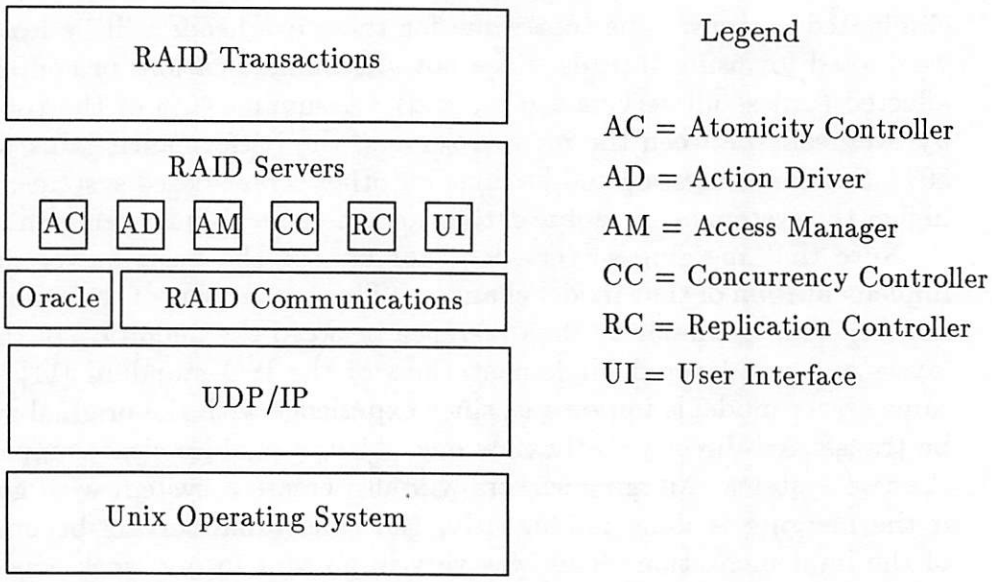


Figure 1: The RAID hierarchy.

simultaneously), and sequence number (several servers of the same type may be running on one site). This provides a natural way to send a message to another server and creates location transparency in the server codes. A server need not know the physical locations of other servers, only their RAID addresses. This had important implications when merging the server codes. An oracle server registers RAID servers and their UDP addresses as they begin execution and distributes this information to other servers. The information is stored by the communications routines and used to translate between RAID addresses to UDP addresses when messages are sent and received.

Figure 2 shows the pattern of server communications in RAID. Each box in the figure represents a RAID server. Arrows represent service requests from one server to another (some arrows represent more than one service request). Unboxed server names represent servers on other sites. The roles of the servers in the RAID system are

- **User Interface (UI)**: a front end invoked by the user to process relational calculus queries.
- **Action Driver (AD)**: accept a parsed query from the UI, format the query as a transaction (read and write actions), and execute the transaction.
- **Access Manager (AM)**: provide write access to the local database, ensuring that updates are posted atomically to stable storage.
- **Atomicity Controller (AC)**: manage the two commit phases of transaction processing to ensure that a transaction commits or aborts globally.

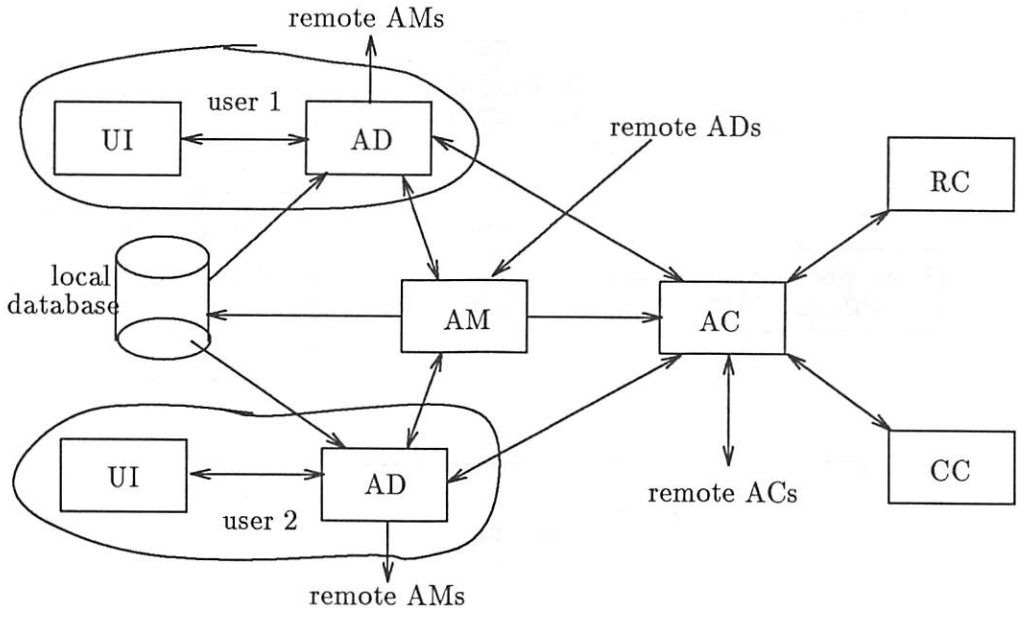


Figure 2: The conceptual organization of a RAID site.

- **Replication Controller (RC)**: maintain consistency of replicated copies of the database in the event of multiple site failures.
- **Concurrency Controller (CC)**: check whether a transaction history is locally serializable at a given site.

One of the main goals of the RAID system was to provide modularity and reconfigurability to allow experimental studies into new methods of distributed processing. This was accomplished by making each server on a site reside in a different UNIX process. The servers communicate with each other using the RAID communications routines. Because of the location transparency provided by those routines, the site and its servers are not tied to any particular host on the network. The design is very attractive because of its adaptability, but the performance of the system suffers because of high communication costs and excessive context switching between the multiple processes competing for CPU time. The study in [2] shows that only a small fraction of the wall-clock time used on a given transaction is directly attributable to server processing; the rest must be attributed to system overhead. We therefore investigated alternatives for reducing this overhead.

3 The New RAID Structure

Our decision to reduce system overheads in RAID by merging conceptually separate servers into one process forced other choices. Merging servers did not change the conceptual organization of RAID from the hierarchy of Figure 1 and the communications of Figure 2. Only the packaging of servers into processes changed.

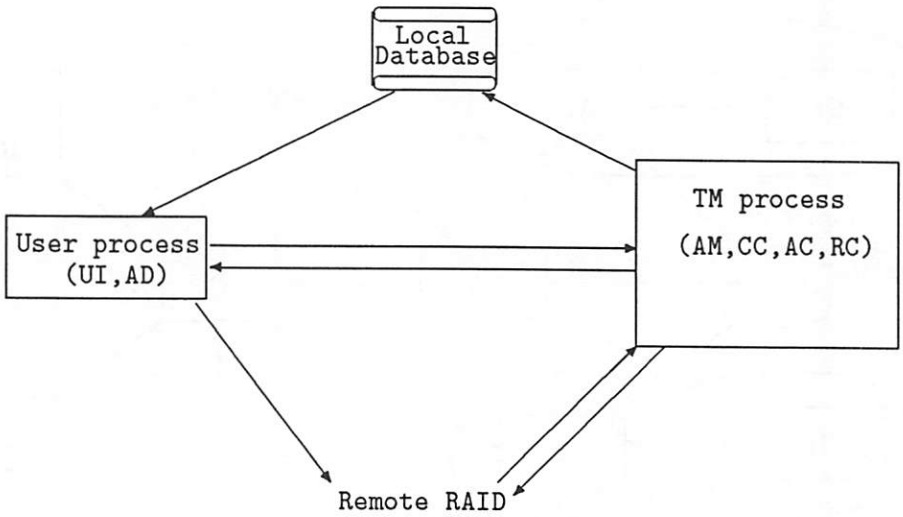


Figure 3: The physical organization of a RAID site.

We decided to merge the AC, AM, CC, and RC into one process, the *Transaction Manager* (hereafter referred to as the TM). The UI and AD were merged into the *User* process. This new organization of a site is shown in Figure 3. The only part of this design that was peculiar to RAID was the choice of servers to merge. The general strategy of merging servers can be applied to any server-based design. It is profitable whenever the system and communications overheads are high.

The division of servers between the TM and User processes was chosen for pragmatic reasons. The AC, AM, CC, and RC servers at a RAID site execute as long as the site is up, while the UI and AD appear and disappear as users come and go. It does not make sense to package permanent servers like the AC with temporary servers like the UI because the server lives are so different. The UI and AD are very closely associated with each other, so packaging them together was a natural decision. It is less clear that having all four servers packaged in one TM process is an advantage, since several transactions might be processed concurrently if the servers were in different processes. This would be possible if there were long latencies in the servers (as there are in the AM) or if the processes could run in parallel (for example, if RAID were ported to a multi-processor). For future investigation of these possibilities, we designed the TM to be configurable at run time with any combination of the four servers. For example, the AC, CC, and RC could be grouped together, with the AM being run in a separate process. This allows us to test and compare various configurations of servers in processes directly.

Several options were open to us for how the servers would be merged.

- *Implement servers as subroutine libraries in the same process.* The major advantage of this approach is that it is very fast, since it can completely

avoid copying data. Thus, the only cost of communicating with a server is a subroutine call, which is on the order of microseconds. The disadvantage is that not all servers act as subroutines. The AC, for example, often can run asynchronously with its caller, so it would be a poor candidate for a subroutine library. If the AM used asynchronous disk writes, it would also be a poor choice as a subroutine. These considerations kept us from using RPC as the communications model in our original server; they now are an argument against subroutine libraries. Because of these problems, we did not use this approach for the TM. In the User process, the disadvantages are much less severe than for the TM. The AD does operate as a set of subroutines, and the User process is currently being written to implement the AD as a subroutine library.

- *Call server routines directly from the communications routines on internal messages*, and use the same interface for messages to internal and external servers. A server wishing to communicate with another server would call the appropriate communications routine. If the message destination were in the same process, the server routine would be called; otherwise, the message would be sent via UDP. This approach has the advantage that server codes need not be rewritten, and it is almost as fast as the first alternative. It has the same disadvantages as the previous approach, however, and it severely violates the layering hierarchy of RAID (Figure 1). Routines at one level of the RAID hierarchy should only call routines at a lower level; thus, the communications routines should not call server codes. Such calls would compromise modularity by making the RAID communications routines less general. Since we give modularity a high priority, we rejected this option.

- *Copy internal messages to a queue*, and use the same interface for messages to internal and external servers. The use of the communications routines is similar to the last alternative. Like that alternative, it requires no changes to the server codes. The problem of servers not behaving as subroutines does not apply here, however, nor is the layering scheme violated. The major disadvantage of this method is that it is not as fast as simple subroutine calls, since it requires copying data. (Enqueuing the data without copying is not feasible, since the data can then be overwritten or even deallocated before the message is received.) This is the method that we used to write the TM.

If the Raid system were designed with a remote procedure call model of communications, it would be relatively straightforward to merge servers by changing RPC calls to local procedure calls. This would be particularly easy if the two types of calls were syntactically alike, as in Emerald [4]. Because Raid uses the more general message-based communications system, however, the merging required more work. Coverting Raid to an RPC model might have reduced this work, but one of our goals was to keep the original design as far as possible.

4 Implementation

This section describes the changes made to the different parts of the RAID code to implement the design in Section 3. The next two sections will describe separately the implementation details of the TM and User processes.

4.1 Implementation of the Transaction Manager (TM)

Most of the work involved is related to this part of the RAID software. Changes had to be made to the communication package, the RAID server code, and the system interface. The next three sections describe all these changes in detail.

4.1.1 Changes to the oracle and communications library

Two basic considerations drove most of our changes to the communications library:

- Multiple servers can now be present in the same process. This was not true under the original implementation; in fact, parts of the communication library depended on having a one-to-one correspondence between processes and servers.
- Internal communication (i.e. messages between servers in the same process) must be efficient. Using UDP to send a message from a process to itself is wasteful.

The first consideration forced us to use a more general addressing translation than the old version of RAID used, while the second forced us to add more code checking for the internal communication case. We will describe the latter change first.

Shared queues are now used for communication among servers residing in the same process. The sending routine, SendPacket, now enqueues the messages if the destination is internal to the process and use UDP otherwise. This requires just one test of the destination address and a short section of code putting the message on the queue. The receiving routine, RecvMsg, now checks for both internal and external messages when called. Priority is given to internal messages by first checking whether the queue is empty. If an internal message is found, it is returned immediately. Otherwise, RecvMsg listens at the UDP socket for an external message. We chose this priority because internal messages are more likely to be related to the currently active transaction; also, testing the queue is faster than listening at a UDP socket.

It is important to note that these changes were only made to the internals of the SendPacket and RecvMsg routines, not to their interface with the rest of the program. Because this is true, the packaging of servers into processes is transparent to the servers themselves. A server can use the same procedure to deliver a message to either an internal server or an external server. Similar techniques could be used

in any server-based system that used virtual server addresses. In general, this allows the servers to be merged making few, if any, changes to the servers themselves.

Having merged servers causes an interesting problem when receiving messages. When a single server resides within each process, there is no need to check which server should receive the message since there is only one possibility. With merged servers, the target server for a given message is not always known. Some way must be available to determine the correct server. Our solution was to include the recipient's RAID address in the message header and have the *receiving* routine read it. We added a new routine, RecvMsgAddr, that reads the destination field and returns it via pointer parameters. The sender's RAID address was already included in messages in this way to facilitate return messages. For backwards compatibility, we kept the old routine, RecvMsg, that ignores the destination.

One minor change was made to the oracle. As part of its initiation, each RAID server registers with the oracle and requests addresses of other servers from the oracle for use in future communications. Under the old RAID implementation, there was a one-to-one correspondence between servers and UDP sockets. Clearly, this is inefficient if several servers are in the same process and can share a socket. The new system allows more than one server to register using the same socket. All RAID communications are sent to that socket; the destination address field is used to dispatch the message to the correct server. The new initialization also excludes internal servers from the list of oracle requests, since their location is already known.

4.1.2 Changes to the server codes

The main structure of each individual server was a main loop that continuously received messages and called the appropriate routines to handle the requests. Figure 4 shows the pseudo-code for the AC as an example. With all the servers merged together, there is only one main loop for all servers, shown in pseudo-code in Figure 5. Messages to the process are demultiplexed to the appropriate server using the destination address field. The server processing routines are unchanged. This new design implies that the servers will execute synchronously. We discuss the loss of concurrency from this decision in Section 6.

4.1.3 Changes to the system interface

We decided to design the TM so that it could be configured with various combinations of merged servers, rather than always containing the AC, AM, CC, and RC. The intent was to allow experimentation with different combinations of merged servers, either to optimize the grouping or simply to observe performance effects. To allow such reconfiguration, we decided to load all the compiled servers into one program and use command line options to instantiate the appropriate servers within the TM. In addition to the new program, we wrote a new version of the RAID "start-site" shellscript. This shellscript starts the servers for an entire site. Formerly, it invoked separate programs for each of the servers; now it only invokes the

```
main( )
{
    InitializeAC( );
    while ( TRUE ) {
        MsgType = RecvMsg( MsgBody, &RAIDAddr );
        ProcessACMsg( MsgType, MsgBody, RAIDAddr );
    }
}

ProcessACMsg( MsgType, MsgBody, RAIDAddr );
{
    switch ( MsgType ) {
      case AD_REQUEST:
        ProcessADRequest( MsgBody, RAIDAddr );
        break;
            :
      default:
        ReportError( "Unknown messge type" );
        break;
    }
}
```

Figure 4: Pseudo-code for AC server.

```
main( )
{
    InitializeAll( );
    while ( TRUE ) {
        MsgType = RecvMsgAddr( MsgBody, &SendAddr, &RecvAddr );
        switch ( RecvAddr.ServerType ) {
          case AC_TYPE:
            ProcessACMsg( MsgType, MsgBody, SendAddr );
            break;
          case AM_TYPE:
            ProcessAMMsg( MsgType, MsgBody, SendAddr );
            break;
          case CC_TYPE:
            ProcessCCMsg( MsgType, MsgBody, SendAddr );
            break;
          case RC_TYPE:
            ProcessRCMsg( MsgType, MsgBody, SendAddr );
            break;
          default:
            ReportError( "Unknown server type" );
            break;
    }
}
```

Figure 5: Pseudo-code for Transaction Manager.

TM. Some new options were also added to the shellscript to control the grouping of servers into processes. The default configuration is to have all servers running in one process.

4.2 Implementation of the User Process

The UI and AD in the original RAID implementation were written very early in the research project, before the RAID communications library was designed. Because our priorities were in the distributed processing aspects of RAID, the UI and AD were never thoroughly integrated with the rest of the system. In particular, they do not exploit the location independence offered by the RAID communications routines. This made merging the servers into the User process more complex than the TM process. We report here on a first attempt at that merging.

In the original version, the UI forked an AD process and communicated with it using UNIX pipes. The UI then forked another program to parse the user transaction, read that program's output from another pipe and copied it to the AD's "read" pipe. The UI read the final result from the AD's "write" pipe and copied it to standard output. The AD *continuously* read from one pipe, processed the transaction sent, and wrote the result to the other pipe. We merged the UI and AD servers into one process to avoid the UI-AD pipe. The AD is now a subroutine that takes its input directly from the parser output and writes its result directly to standard output. This is a slight improvement on the original design, but is still inelegant. A new version of the User process is currently being written which merges the servers by implementing the AD as a subroutine library. We hope this will make the new program both more efficient and more maintainable.

5 Measurements of Original and New RAID Systems

We made several measurements on the new RAID system to compare it with the old version. The first set of measurements compares the costs of the new internal communication routines. Since the implementation of the new system involved creating two independent merged servers (the TM and the User process), two additional sets of measurements for transaction processing time were collected. One was gathered with only the TM being active, i.e. the UI and AD were still separate processes. The other set was collected with both merged servers being active. The transaction benchmarks and measurements are as described in [3,2].

5.1 Measurements of Communication Times

In order to explain the performance times obtained from the new design, some measurements were collected on the new communications used. Table 1 compares the time to send messages of different sizes using the old RAID communication

Bytes:	64	512	2048	8192
External (using UDP)	11.9	20.1	46.7	153.3
Internal (using queues)	1.1	2.7	8.3	30.7

Table 1: Round trip external and internal communication times by packet length (in milliseconds)

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.3	0.3	0.4	0.4
select eleven tuples	0.4	0.4	0.4	0.4
insert twenty tuples	0.6	0.6	0.8	0.8
update one tuple	0.4	0.4	0.4	0.4

Table 2: Transaction execution time for original RAID system (in seconds).

routines (built on top of UDP) with the time for the new internal message queues. Communication between servers in different processes still uses UDP; the times for those messages is essentially unchanged. Our new routines, however, require 80 to 90 percent less time for internal messages.

5.2 Measurements of Transaction Execution Times

Table 2 (taken from [3]) shows the times taken for transaction processing for several database queries using the original RAID system. The times include only the cost of committing the transaction; cost of parsing the query is ignored. Table 3 shows the times for the same queries using the TM process, but with the old UI and AD. The configuration for the TM packaged the AC, AM, CC, and RC together. For one site, the new system is 33 to 55 percent faster than the old. For four sites, the figures show a speedup of 7.5 to 20 percent. The lower speedup for multiple

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.18	0.19	0.19	0.32
select eleven tuples	0.18	0.19	0.22	0.34
insert twenty tuples	0.40	0.40	0.45	0.64
update one tuple	0.22	0.22	0.22	0.37

Table 3: Transaction execution time for RAID system with new TM and old UI and AD (in seconds).

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.12	0.13	0.13	0.25
select eleven tuples	0.13	0.16	0.17	0.31
insert twenty tuples	0.35	0.37	0.37	0.57
update one tuple	0.16	0.17	0.18	0.32

Table 4: Transaction execution time for RAID system with new TM and User Processes (in seconds).

sites is to be expected, since they require external communications for site-to-site messages.

These improvements approximate our expectations based on the improvement in message-passing times and the numbers of messages converted from external to internal. The amount of time saved for each transaction with the new design is proportional to the reduction in the number of UDP messages. The first two rows of the table only involve reading items from the database, which saves 4 UDP messages out of 6 total messages (in the single-site case). Thus, if the average packet length were 512 bytes, Table 1 shows that the time savings from communications alone would be $4\times(20.1-2.7) = 69.6$ milliseconds. This is just over half the actual speedup of 120 milliseconds for selecting one tuple on one processor, which is reasonable for such a simple estimate. The last two rows involve writing to the database, saving 3 more UDP messages of an additional 5 messages (again, in the single-site case). Here, the expected communications speedup is $7\times(20.1-2.7) = 121.8$ milliseconds, again in reasonable agreement with the experimental data. In both cases, more messages are needed in the multiple-site case to coordinate with remote sites; these must be sent via UDP, so no further speedups can be expected.

Table 4 shows similar transaction execution times in RAID systems containing both a TM and a User process. The speedups range from 42 to 68 percent for one site and 20 to 38 percent for 4 sites.

Almost all of the improvement between Tables 3 and 4 is due to fixing a performance bug in the original AD (file descriptors were being closed without first being opened). We cannot credit this improvement to our merged server design; it was simply a case of stumbling over an old bug and fixing it. If the time for parsing the transaction were also included in the above numbers, there would be a further time savings for the User process. This is because one level of indirection (the UI-AD pipe) has been eliminated.

6 Analysis and Conclusion

We have accomplished our goals in this project: to improve the performance of the RAID system while maintaining its modularity. We achieved this by creating

a difference between the virtual system architecture (the communications links in Figure 2) and the actual implementation (the process packaging in Figure 3). In addition, we think that future changes to the RAID system can be made within our "virtual server" implementation. This idea is currently being tested, as some changes to the virtual system architecture are being implemented at the server level without modifying the communications routines or the TM main loop. Similar merging of conceptual servers can be done on any server-based system. If implemented carefully, it can eliminate overhead without sacrificing modularity or redesigning the conceptual system. In fact, it is possible to do the merge without changes to the server routines themselves.

While merging all four servers certainly minimizes communications cost, it also forces the servers to run synchronously. This may be a disadvantage if many transactions are run concurrently or if RAID is ported to a multi-processor machine. Since the RC and CC only communicate with their local AC, the best configuration on a single processor should include all three servers in one process. Concurrency is not an issue in this case, since the RC and CC cannot run in parallel (for the same transaction) and only one process can be executing. The AM also communicates with the AC so we do save one UDP message by including the AM in the same process. However, the AM's main job is to write data to disk. Since we only use synchronous I/O, it may be an advantage to run the AM in a separate process even on a uniprocessor to avoid blocking the entire TM. If the machine running RAID is a multi-processor, we may have to redesign the system to exploit its concurrency. As a simple example, it would be useful to have each server working on a different transaction in parallel on such a system. We are currently experimenting with different configurations of the TM to determine how much effect this loss of concurrency has.

We close with some lessons we've learned that we think are applicable to other distributed systems:

- Modularity is important to accommodate new algorithms and techniques in research systems.
- It is possible to have both modularity and reasonable performance.
- The modularity of the implementation (process packaging) need not reflect the modularity of the design (servers).
- Overhead in a distributed system can be reduced by collecting conceptual servers into a single physical entity.

Acknowledgements

We would like to thank Dr. Jim Gray of Tandem Corporation for suggestions that led to the merged server design. John Riedl was helpful in the initial stages of our

design. We thank all all the students at Purdue University and the University of Pittsburg who have contributed to RAID over the years. Our work would not have been possible without theirs.

Bibliography

References

[1] Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder. Implementation and measurements of an efficient communication facility for distributed database systems. In *Proceedings of the 5th IEEE Data Engineering Conference*, Los Angeles, CA, February 1989.

[2] Bharat Bhargava and John Riedl. Implementation of RAID. In *Proc. of the 7th IEEE Symposium on Reliability in Distributed Systems*, Columbus, Ohio, October 1988.

[3] Bharat Bhargava and John Riedl. The RAID distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-12(12), December 1986.

[5] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[6] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[7] Barbara Liskov and R. Scheifler. Guardians and actions: Lingustic support for distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[8] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49, August 1986.

[9] J.B. Rothnie et al. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5:1–17, March 1980.

[10] *Guide to the CAMELOT Distributed Transaction Facility.* Carnegie Mellon Computer Science Department, 0.98(51) edition, May 1988.

[11] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

Making *libc* Suitable for Use by Parallel Programs

Julie Kucera

CONVEX Computer Corporation
Richardson, Texas
kucera@convex.com

ABSTRACT

The development of the CONVEX parallelizing C compiler, required changes to the standard C library, *libc*, to make all of its routines suitable for use by parallel programs. Many routines in *libc* were not reentrant and therefore could not be executed in parallel by multiple execution streams of the same process. This paper describes the project undertaken at CONVEX to utilize semaphores or other means to protect the data used within *libc* from concurrent access.

Motivation

The motivation for the "semaphoring" of *libc* originated with the compiler development group at CONVEX. A parallelizing C compiler was under development to accommodate the users of our multiprocessor machines. Although the automatic parallelization the compiler was to perform did not include parallelizing across function calls, new directives were added that would allow C programmers to specify regions of code that should be executed in parallel. Once these parallel capabilities were provided to our customers, they might not realize that calls to *printf*, *malloc*, and other common *libc* routines would not always work in parallel. Even if our customers knew which routines were not reentrant, there is some work involved in actually guaranteeing mutually exclusive access to all of the critical data or code sections. For these reasons, we felt that it was necessary to provide reentrant library routines to prevent the unpredictable errors that could result otherwise.

Overview of the CONVEX Architecture

Before detailing the project itself, it is helpful to explain a little bit about the CONVEX architecture and to define some terms [1].

ASAP® - Automatic Self-Allocating Processors, a unique architecture designed by CONVEX. A cornerstone of ASAP is the communication register, which allows CPUs to seek out and execute the next piece of work as soon as possible. The hardware claims work without operating system intervention, other than the organizing of jobs in the run queues based on priority.

communication register - A high-speed register used for communication between the threads of a process. Threads communicate by sending and receiving data through the communication registers. A hardware-maintained lock bit is associated with each register to guarantee mutually exclusive access to the register.

CPU - One physical processing unit, sometimes referred to as a "head.". Each CPU in the configuration operates independently as a 64-bit CONVEX supercomputer, including maintaining its own memory cache. The CONVEX C200 series of computers contain from one to four CPUs.

multiprocessor - A machine that contains more than one CPU; also called a multi-headed machine. The processors are symmetric (i.e., there is no master-slave relationship) and tightly-coupled.

process - A collection of one or more execution streams within a single logical address space.

semaphore - A shared data structure used to synchronize the actions of multiple cooperating processes or threads. The two primitive functions that operate on semaphores are *wait*

and *signal*. The *wait* function is normally executed before entering a critical section. The function waits until the semaphore, or lock, can be exclusively acquired. The *signal* operation is called to atomically release a semaphore and is executed upon exit from a critical section. In this paper, the terms *acquire* and *release* are used as synonyms for the standard semaphore operations *wait* and *signal*.

thread - An independent execution stream that is fetched and executed by a CPU. A process is made up of one or more threads, each of which can execute on a different CPU.

thread memory - Memory that is allocated to a single thread and is not shared among the threads constituting a process. Thread-specific, or thread private, data results in the same virtual address in different threads referencing different physical memory locations.

Problem Statement

There were several problems involving data consistency that needed to be solved in our effort to parallelize *libc*. Perhaps most evident were the numerous data structures scattered throughout the library routines that resided in **static** or otherwise global storage. Each such data element needed to be protected from concurrent access. But there were also more subtle instances of shared data states that, left untouched, would easily lead to corrupted data or wrong results. For example, contained in the *stdio.h* include file were a few simple *#define* macros that manipulate global data. Also, the *getwd* routine changes the current working directory to the parent directory, and if that critical code section was left unprotected, simultaneous calls to this routine could obtain different answers.

Other considerations, such as the packaging and backward compatibility of the new routines, will also be briefly discussed.

Overview of the Project Strategy

The primary focus of the project began with protecting all of the static and global variables. The methods used to obtain this list of data will be described later. We had two ways to protect global data from corruption by concurrent access — using semaphores and thread private memory.

One method was to place semaphores around access to the shared data, forcing each thread to gain exclusive access to the data before modifying it in any way. But there were some cases in which semaphores would not be useful. For example, some routines return a pointer to static data defined within the routine. The caller expects the contents of the data to remain constant until the routine is called again. Adding semaphores within these routines would be pointless. One possible solution for this problem is to modify the library interfaces. The caller would pass a pointer to a buffer on his own stack that would be filled in by the callee. This approach has the serious drawback of incompatibility with existing programs. With thread memory in the CONVEX architecture, a more elegant solution was possible. This solution was to place the data in thread private memory, so that each thread would have consistent data that remained intact between subroutine calls within that thread. These changes are invisible to the user and are completely backwards compatible. Thus, the second method undertaken undertaken to protect static and global data was to use thread private memory.

Conventions for Thread Memory. Before outlining the general changes that were made to the routines, let us expand on the steps taken to move data into thread memory. Because our C compiler did not provide a method of declaring within a C source file that a particular variable should be placed in thread memory, the following standard procedure was developed for smoothly achieving this movement.

First, each file *filename* that needed to use thread memory had its declarations for the thread memory variables placed in a separate file named "*filename*_thread.thr." This naming convention linked this separate file with the routines the data was used for, and it allowed compilation with a special ".thr.o" rule in the makefile. These ".thr" files are first compiled into assembly language to have the space allocated and aligned properly, then *sed* scripts are used to transform the (global) data/bss sections into thread data/bss. The resulting ".s" file is assembled into a ".o" file to be placed into the library.

Since the formerly **static** data variables have had their scope increased from a single source file (or routine) to the entire program, their names needed to be changed. The string "*filename*$" was prefixed to each variable name, where *filename* is the name of the original source file, as

above. The intention was to reduce the likelihood of conflicts with other variable names.

The variables that were placed in thread memory were obviously still needed in their parent files. We chose to "declare" them as items to be stored in thread memory by using the word **thread** as if it were a C storage class. The C preprocessor was then used to define "thread" as "extern," thereby providing the proper declaration. This convention was followed in hopes that a future version of the compiler will recognize **thread** as a keyword, eliminating the need for the special processing, which was designed to be easily removed.

Determining and Making the General Changes

Protecting Static Variables. Before proceeding, we needed to identify all of the variables that had the potential of being corrupted. To locate all of the static variables, a search for the word "static" was performed on all source files that comprise *libc*. This process located 88 source files, some containing as many as 20 static variables that each had to be fully examined for usage to decide which protection approach to take. Listed below are the different purposes we discovered for using static variables, followed by the action we found appropriate in each case:

1) the word "static" only applied to a function
   —> no action
2) in a read-only table of values
   —> added comments to the code stating that it is read-only
3) in a routine that cannot be called in parallel because it had a *vfork/exec* in it (e.g., *popen*, *syslog*)
   —> no action
4) as a static global for convenience (accessibility)
   —> passed it as an argument to the routines that needed that variable
5) as a variable that was not necessary
   —> used a *#define* instead (e.g., a fixed filename)
6) the static location was used to return data to the caller (e.g., *getpwent*)
   —> put the variable into thread memory, changed its name (as described earlier) in all instances and created a new "*filename*_thread.thr" file containing all such variables in the file
7) the variable was used (read and written) internally as global data (e.g., the buffer lists within *malloc/free*)
   —> surrounded all accesses to the variable with semaphores, so that only one operation would occur at a time

Protecting Global Variables. To find these variables, we ran *nm* on *libc*, then used *grep* to find the various forms of global (external) data, namely D (data), C (common), S (initialized common block), and B (bss). There were about 30 variables found in this manner. Using *nm* was helpful again in finding all the routines that referenced each one of these variables (e.g., for the *_iob* array used in standard I/O routines, nearly all *stdio* files needed to be changed). The following actions were taken for the given cases:

1) as a read-only table of values
   —> added comments to the code stating that it is read-only
2) only used as a temporary variable in each routine
   —> declared it as a local variable in each routine that needed it
3) data needed to be used by the caller (e.g., *errno*, *optarg*)
   —> moved into thread memory, this time *not* changing its name, because the name was previously global and intended to be referenced by user code
4) only one value for an entire program (as opposed to one value per executing thread) made sense for this read-write datum (e.g., *curbrk*)
   —> added semaphores around all accesses to the data variable

More Quick Tips. Routines that have multiple **return** statements must be certain to release any semaphore they hold before each **return**. Also, further modifications were required for the functions that returned values which needed to be surrounded by semaphores. The calculation of the result to be returned must be stored in a local variable while the thread owns the semaphore. After release of the semaphore, the local variable is returned. This approach works because each thread is allocated its own stack, and local variables are stored on the stack and therefore guaranteed to belong to only one thread.

The Semaphore Routines

Once the need for semaphores was established, it was necessary to determine how the acquire (*wait*) and release (*signal*) primitives must work.

The Problems with (Software) Signals Since there needed to be several semaphores added throughout *libc*, the possibility of receiving a software signal while holding a lock had to be considered. Because there is no way of knowing what a signal handler will do (and which *libc* routines it may call), deadlock could occur when a signal handler calls the same routine it just interrupted and tries again to acquire the lock. Perhaps the safest way to avoid such deadlock would be to block all signals while the semaphore is held, unblocking upon semaphore release. This first approach has the potential problem of having one thread block signals while another thread is changing the signal mask. Then, when the thread that blocked signals while in its critical section tries to restore the signal mask to its old value, the changes made by the other thread will be lost. Another drawback to selecting the signal-blocking method is the overhead of adding *sigblock* calls to the entry to and exit from every critical *libc* routine.

The second option we considered was to make all lock acquisitions nestable. In the case of nested calls to routines that need to acquire the same semaphore, an attempted acquisition will only fail if the lock is held by another thread. This adds a small amount of overhead in keeping up with "who" (meaning which thread) currently holds a lock, and a count of how many times it has been "acquired" so that we know when to actually release it. This second option would behave as the library routines always did in terms of potential signal handlers (that is, there is no mechanism in the *libc* routines that protects against, for example, a *malloc* call within a signal handler for a signal that interrupted a *malloc* call already in progress — the possibility of corrupting *malloc*'s internal data structures has always existed).

We arrived at a consensus to make all lock acquisitions nestable. There was not a reliable way to ensure that the "right" things always happened if we chose to block signals (because of the possibility of another thread changing the mask to allow some signals when we were trying to block all signals). Another factor in our decision was the fact that nestable locks were already a desirable trait for some semaphore instances (such as those routines that call other routines that use the same semaphore, for example *_filbuf* and *fflush* in the *stdio* sublibrary).

Writing the Semaphore Routines. After agreeing that we needed to have nestable semaphores, we wrote the assembly routines that would perform the acquiring and releasing. This task was particularly difficult because these routines run in user space, not within the context of the kernel. We needed to guarantee that the necessary operations were atomic. We didn't have the capability of ignoring interrupts and context switches until we were ready for them. A simple test-and-set binary semaphore was insufficient because of our need to store the current owner of the lock as well. We also thought it was necessary to keep a count of how many times the lock had been acquired so that we would know when it could actually be released. Since we did not have a way to atomically store two values (one for the owner and one for the count), there was always the possibility of getting interrupted in the middle of the lock operation, which would render it useless.

The design of synchronization primitives to meet all the requirements was quite complicated — until we realized that the count field was unnecessary. All we needed to know from the lock-acquire routine was whether the lock was actually obtained in that instance of the call, or if we were allowed to continue with the lock because we already held it. So, the acquire routine was written to return one if the lock was really acquired, and return zero if this was a nested call. The caller of the semaphore routines was responsible for checking this return value and deciding whether or not to call the release routine.

C1 vs. C2 Class Machines. Not all of CONVEX's machines are multiprocessors. The first generation machine (the C1) is not capable of running parallel programs [2]. New instructions were added to the second generation machine (the C200 series) specifically for creating, joining, and synchronizing among threads. The differences in machine types introduced several issues. First, the semaphore routines we wrote utilized a synchronization primitive that doesn't exist on C1 machines. Also, the overhead involved in acquiring and releasing semaphores is unnecessary if you are certain you will never need it. Finally, we didn't want to ship and support two separate libraries, one for C1s and one

for the C200 series.

We solved all these problems with the introduction of a global variable (called "use_libc_sema") that would be set at startup time to indicate whether or not any semaphores should be used. This is determined by both checking the machine type of the machine we are running on and by examining the potential to go parallel. Our assembler and loader set certain flag bits in the executable, which indicate if the program contains any parallel instructions (the kernel then can use this information to disallow execution of such a program on a C1). We know that if there are no such instructions, then the program will never enter parallel mode, so no locking for exclusive access will need to be done, even if it is running on a multi-headed machine. Similarly, even if there are parallel instructions in the executable, but we are running on a machine which only has one CPU (such as a C210) — there is a system call that provides such information about the system — there is no need to add the overhead of semaphore usage.

The modifications that involved thread memory will work correctly on both classes of machines, because thread memory on a C1 is treated like all other memory.

Using the Semaphore Routines. Each use of the semaphore primitives must therefore first check the value of the "use_libc_sema" flag before calling the acquire or release function. Additional processing around the semaphore functions is also needed to take care of detecting when to do the actual release since the lock acquisitions are nestable. For these reasons, macros were written and then used in all semaphore instances. These macros are shown in Figure 1, along with an example of their use:

What about *stdio.h*? An easy oversight might be include files that define macros that modify global variables, such as the *putc* and *getc* macros in *stdio.h*. This introduces yet another problem. How can one semaphore around data within a simple macro that needs to return a value? A function call is the best place to encapsulate all that needs to take place, but we don't want to unnecessarily pay the overhead of a function call for these commonly-used macros. So, we modified the *putc* and *getc* macros to check the value of "use_libc_sema," calling their function-call counterparts *fputc* or *fgetc* only if necessary.

```
/* macros for using the semaphore routines in libc */
#define Uses_Sema       int release_sema

#define Sema_Acq(s)     if (use_libc_sema)
                            release_sema = _lck_acq(s)

#define Sema_Rel(s)     if (use_libc_sema &&
                            release_sema)
                                _lck_rel(s)

/* example usage of the above macros */
extern Libc_Sema    _iosema[_NFILE];

fgetc(fp)
FILE *fp;
{
    Uses_Sema;      /* define var used in sema macros */
    int c;          /* value of macro to return */

    Sema_Acq (&_iosema[fp->_sema_ndx]);
    /* use macro version since can sema around */
    c = M_getc(fp);
    Sema_Rel (&_iosema[fp->_sema_ndx]);
    return(c);
}
```

Figure 1

Another issue is brought up in *fgets*, which calls *getc* repeatedly. We don't need to acquire the semaphore each time we call *fgetc* since we already have the semaphore for the duration of the *fgets* routine. We created definitions in a local include file of *putc* and *getc* that are always macros, called *M_putc* and *M_getc*. The *M_getc* macro was also used in the *fgetc* function while the semaphore was held, as shown in the example in Figure 1.

There was also the issue of where to put all of the semaphores that were needed for each possible FILE * (struct _iobuf). We originally added a field to the end of the existing _iobuf structure because it made sense to keep the semaphore with the rest of the file-pointer-specific data. Unfortunately, that didn't work. In fact, everything that included the original *stdio.h*, but was relinked with the new standard I/O routines, failed miserably on all I/O that was written to a file other than *stdout*. Since the array element size had changed, any reference beyond the first element in the "_iob" array (almost always *stdout*) had been calculated differently by the existing program and the new library routines. This discrepancy caused the program's supposed file pointer to actually point to the middle of a different FILE structure (in the eyes of the library routines), which was disastrous. A complete recompilation of the source files that used *stdio* would fix the problems, but that was an unrealistic alternative. The resolution was to create an additional array of semaphores, one per FILE structure, and provide a way to quickly associate each semaphore with its corresponding

FILE. Although the file pointer passed to all standard I/O routines could easily be used (in combination with the base address of the FILE structure array and the _iobuf structure size) to calculate the proper index into the global semaphore array (*_iosema*), it was more efficient to store this index (_sema_ndx) directly in the _iobuf structure during the call to *fopen*. There was a byte-size hole after the **unsigned char** field that holds the file descriptor (_file), so the _sema_ndx field could be added without enlarging the _iobuf structure. (Refer back to Figure 1 for a clarification of this association between file pointer and semaphore.) Should this utilization of the extra "unused" byte in the _file field cause problems in the future, we can simply modify the *stdio* routines to calculate the correct semaphore index since corresponding _iob and _iosema array entries will have the same index. But we don't wish to add these extra calculations until we have to.

Testing the Solutions

First Ending. Before testing the new parallel features, we wanted to confirm that no problems were introduced in single-threaded mode. Something as fundamental and universally used as *libc* warrants a considerable testing effort. All of our utilities were recompiled and linked with the new *libc*. CONVEX has a fairly large test development group which writes automated tests for our software, so there were already several hundred existing tests to run against these recompiled utilities. These tests gave us the confidence that existing functionality had not been changed.

While the library modifications were taking place, a member of the test group was writing tests that would perform certain library calls in parallel, synchronizing the threads immediately before invocation of the library routine. Many of these tests were designed to run against the *stdio* routines. For example, one test made simultaneous calls to *fprintf* with different strings (all **A**s for thread 0, all **B**s for thread 1, etc.), but the same (shared) file pointer. It then verified that, although the order of the strings in the output was non-deterministic, none of the strings were corrupted by characters from a different string. That is, there were to be no **A**s and **B**s scrambled together. Another critical and common routine that we wanted to test thoroughly was *malloc*. Within a few hours, all of the parallel tests passed repeatedly... on a two-headed machine, anyway.

At the time of our testing, our four-headed machines were limited in number, with most of the time on them being taken up by marketing benchmarks. We couldn't consider the project completely finished until we ran the tests on a C240. Once machine time became available, we ran the tests. The first test run was successful! But, repeating the tests 20 or 30 times resulted in a few periodic failures!

At first we thought we missed a semaphore or two (after all, there are several hundred *libc* routines). The possibility that it could be a hardware problem also ran repetitively through our minds, since the failures never occurred on a C220 (the two-processor machine). Searching through the code to the failing routines brought no enlightenment. We ran the tests under *adb* over and over again. Ever notice how bugs are afraid of debuggers? They don't like to appear in those circumstances!

We were finally able to prove (after weeks of working in *adb* and narrowing the test cases down) that there were two threads being awarded the lock at the same time! We examined the lock acquire and release routines and determined that the problem must be due to cache inconsistency among the CPUs. We experienced these problems because the primitive we were required to use for the acquire routine did not have an atomic counterpart for the release routine. We needed to be able to clear *both* the owner and a field in the semaphore structure that indicated whether the owner value was valid data. By design, when the semaphore is in use, the data-valid bit is set, and when the semaphore is free, the data-valid bit is clear. The instruction that wrote a value into the owner field of this particular semaphore structure also set the data-valid bit, and the instruction that cleared the data-valid bit did not modify the "value" of the semaphore (the lock owner). We had been working around this limitation by using a simple load instruction to clear all 64 bits in the structure, which apparently was not working. Because it was the release routine that was occasionally causing the inconsistency, it took more than two processors to reproduce the problem. If only one other thread was contending for the lock, it didn't matter that the first thread left the semaphore structure in a less-than perfect state for a brief interlude — the waiting thread would be awarded the lock and enter the critical section with no interference from the other

thread, which had already completed its execution in the critical section.

Rewriting the Semaphore Routines. We held a meeting with hardware, software, and compiler engineers and managers to discuss these revelations. Our only option was to design a new instruction that would provide the functionality we needed in a truly atomic and reliable manner. Since it was certain we were not going to change the design of the machine to have only one memory cache in a configuration, it was evident that we would need hardware support to make atomic the functions we needed to be atomic. A fast-paced schedule was set up to coordinate between all the involved groups to get this done as quickly as possible, because the parallelizing compiler was scheduled to ship that same week! We added a compare-and-swap instruction, as summarized in Figure 2.

```
msync;                  /* wait for active memory  writes */
if (tas(effa.lock)) {
    C = 1;              /* tas successful */
    if (c(effa.data) == Sk<32..63>) {  /* match hi word? */
        c(effa.data) = Sk<0..31>;      /* data = lo word */
        SC = 1;     /* match successful; store occurred */
    } else {
        Sk<32..63> = c(effa.data);     /* ret cur data */
        SC = 0;     /* match failed; store aborted */
    }
    tac(effa.lock);     /* release resource */
} else {
    C = 0;              /* tas failed */
    SC = 0;             /* match failed */
}
```

Figure 2

Below are some notes on the above implementation:

1) *effa* represents the effective address of the semaphore resource structure, where *lock* and *data* are fields of that structure. *sk* represents a 64-bit scalar register, with the upper half containing the value to match, and the lower half containing the new value to place in the semaphore data field if the compare succeeds.

2) Both Address Carry (C) and Scalar Carry (SC) are always modified. Address Carry is set if the test-and-set was successful. Scalar Carry is set if the compare succeeded and the replacement data was stored in the resource structure.

3) If the compare fails, the current data value of the resource structure is put in the upper half of the scalar register.

4) This instruction is atomic.

Using the New Instruction. Figure 3 shows how this new instruction can be used for locks that keep track of the current owner, with the following assumptions:

a2 -Contains a pointer to a memory resource structure. The data field indicates if the lock is held: zero indicates lock is not held; otherwise the value is the ID of the lock owner.

a3 -Contains the new lock owner ID.

We chose to use a combination of the process ID and the thread ID for the lock owner ID. This allowed us to have a *libc* that was suitable for use by multiple threads and multiple tasks (which are not described here), as long as the "use_libc_sema" variable is properly set. A variation of the thread ID (being careful not to use zero) is sufficient for strictly multi-threaded processes.

```
lock:   sub s0,s0        ; clear s0 (we want to match unlock)
        mov   a3,s0      ; set new ID as replace value
3$:     casr.w a2,s0     ; try to acquire lock
        jmpa.f 3$        ; if tas failed
        jmps.t 4$        ; we acquire lock
        mov   a3,s1      ; test to see if we already own lock
        shf #-32,s0      ; position current owner for test
        eq.w  s0,s1      ; test if current owner == new
        jmps.f lock      ; if we don't own lock
4$:                      ; continue

unlock: mov   a3,s0      ; current owner to s0
        shf #32          ; position to match, replace value is 0
5$:     casr.w a2,s0     ; unlock if we own it
        jmpa.f 5$        ; if tas failed
        jmps.f panic     ; we didn't really own the lock
```

Figure 3

Notice that the given routines use spin-waiting for the semaphore locking. The primary reason we used a spin-waiting lock was that we anticipated the contention for each library semaphore to be very low. Also, the amount of time each lock would be held is short enough to have another thread wait on it. A queue of waiting threads per semaphore would add considerably more overhead than we deemed necessary or desirable.

Second Ending. After the new instruction was implemented, we eagerly installed the new microcode on a C240 and started the tests. We ran them all hundreds of times without error.

Summary of Changes

After applying all of the techniques described in this paper, here is a brief summary of the extent of the resulting changes, which account for

about two months of effort:

— slightly modified *crt0.s* (the C startup routine)

— 96 modified C source files

— 2 modified assembly source files

— 10 modified makefiles

— 1 modified include file (*stdio.h*)

— 41 new ".thr" files for thread memory

— 3 new ".h" include files

— 1 new assembly file
(for *_lck_acq* and *_lck_rel* routines)

In addition to the 256 semaphores added for the FILE structures within *stdio*, there were 29 other semaphores introduced during this process. There were also a total of 142 variables defined in the 41 files created for thread memory.

Conclusions

The experience we gained through this project can be used when writing future library routines that have the potential of running in parallel. Many decisions had to be made that would protect backward compatibility. There were some cases that, because of their nature or implementation, could not be efficiently and reliably addressed.

Limitations. There are a few routines that were intentionally not semaphored. For example, *sleep* was skipped because there did not appear to be a way to allow parallel *sleep*s (especially of different time lengths) as well as provide a reliable way to handle any real interval timer that was previously set. This limitation is caused by the implementation of *sleep*, which sets its own interval timer, waits for a SIGALRM signal (temporarily changing its signal handler), then restores the old interval timer. We documented in the manual page that unpredictable results may occur if multiple *sleep*s are attempted in parallel.

The *signal* library call was also left intact. All data used in this routine are kept locally on the stack and all signal-related system calls are already semaphored within the kernel. But, there may someday arise a need to provide a semaphored parallel signal routine that can keep track of the *sigvec* and *sigmask* values to be restored when several signal calls are made concurrently.

Performance. The solutions involving thread memory resulted in a slight increase in overhead for virtual-to-physical address translation, namely that of indexing by the thread ID into an additional level of page table entries. There was also a very small amount of overhead added to all critical sections within *libc* that warranted semaphores. In instances where the process will remain single-threaded, this overhead is merely the cost of a simple **if** statement, approximately an additional .5 seconds per million tests. Where parallelization is possible, the program speedups that can be obtained by executing multi-threaded far outweigh the cost of the semaphore acquisitions and releases. Simultaneously performing many of the common tasks that are provided in *libc* would not have been feasible at all had this project not been undertaken.

Future Enhancements. If the semaphore routines did not depend upon instructions that are only available on our second generation machine, the work done for this project could also be used to synchronize among multiple tasks. There is already a mechanism in place for distinguishing between the same thread IDs of different tasks when testing the current owner of a lock, but there is currently no way to determine the proper initial value of "use_libc_sema." That is, the potential of spawning multiple tasks that will share the semaphored data cannot be determined, like the potential to enter parallel mode can. And, no semaphore routines will be called if "use_libc_sema" is not true. Multi-threaded, multi-tasked processes will have the necessary semaphore protection, but single-threaded, multi-tasked ones will not, because the value of "use_libc_sema" is not set automatically for the user.

With POSIX compliance and ANSI Standard C rapidly being incorporated by CONVEX, many of the variable names used in this project may have to be changed to avoid breaking the rules against namespace pollution.

References

[1] *CONVEX Architecture Reference, Third Edition*, Document No. 081-000050-201, CONVEX Computer Corporation, Richardson, Texas, October 1988

[2] Tom Jones, "Engineering Design of the Convex C2," *IEEE Computer*, January 1989

Revolution 89

or

"Distributing UNIX Brings it Back to its Original Virtues"

François Armand, Michel Gien, Frédéric Herrmann, Marc Rozier

Chorus systèmes
6, avenue Gustave Eiffel, F–78182, Saint-Quentin-en-Yvelines (France)
Tel: +33 1 30 57 00 22, Fax: +33 1 30 57 00 66, E-mail: mg@chorus.fr

1. Introduction

The need to handle distributed computing in a general manner leads us to structure our operating systems functions in a much more modular way than it is done in today's systems such as current UNIX kernels, and to provide facilities for dynamic reconfiguration so the system can be adapted to the variety of configurations needed in a distributed system. When applied to the UNIX kernel, such a "restructuring" leads to same kind of "revolution" that UNIX performed on the operating systems of the 70's, i.e., to extract from the operating system all functions that can better be performed outside, and to leave in the kernel only those generic services that are necessary to provide higher level services, such as high level file access methods, command languages (shell) or system administration functions.

The CHORUS[1] architecture is designed to support new generations of open, distributed, scalable operating systems. It allows the integration of various families of operating systems, ranging from small real-time systems to general-purpose operating systems, in a single distributed environment.

The CHORUS architecture is based on a minimal real-time Nucleus that integrates distributed processing and communication at the lowest level. CHORUS operating systems are built as sets of independent system servers, that rely on the basic, generic services provided by the Nucleus i.e., thread scheduling, network transparent IPC, virtual memory management and real-time event handling.

The CHORUS Nucleus itself can be scaled to exploit a wide range of hardware configurations, such as embedded boards, multi-processor and multi-computer configurations, networked workstations and dedicated servers.

Operating systems (called Subsystems) implemented on top of this Nucleus currently include a UNIX[2] SYSTEM V[Herr88] and the "Emeraude"[Mino88] CASE/PCTE system. Work is currently in progress to implement Object-Oriented distributed Subsystems.[Alve88]

CHORUS–V3 is the current version of the CHORUS system developed by Chorus systèmes. Earlier versions were designed and implemented within the Chorus research project at INRIA between 1979 and 1986. Related work includes the V-system[Cher88] for the message-passing kernel, Mach[Rash87] and[Li86] for the distributed virtual memory, Topaz[McJo88] and Mach[Acce86] for the "threads", Amoeba[Mull87] for the global addressing, and the Bell Laboratories'9th Edition UNIX[Pres86, Wein86] for the uniform file naming.

CHORUS–V3 is written in C++ (and C). It currently supports 680X0 and 80386 based machines, with implementations on networked workstations and servers, as well as multi-processor configurations.

---

1. CHORUS is a registered trademark of Chorus systèmes
2. UNIX is a registered trademark of AT&T

This paper outlines the architecture and implementation of UNIX kernel functions in terms of the CHORUS architecture concepts, based on the CHORUS Nucleus basic services. It focuses on the experiences drawn from it, the resulting benefits for users as well as for systems designers and maintainers, and the issues that still need to be considered.

The next section summarizes the CHORUS Nucleus' basic abstractions and services as described extensively in. [Rozi88] Section 3 outlines the structure of a UNIX Subsystem, in terms of independent cooperating CHORUS servers, illustrating how one can make use of the Nucleus facilities in a UNIX context. Straightforward extensions in the services provided at the UNIX kernel interface level will also be presented. The remaining sections give examples of using the CHORUS architecture in typical system configurations and operating system experiments.

2. The CHORUS Architecture

2.1 Overall Organization

A CHORUS System is composed of a small-sized **Nucleus** and a number of **System Servers.** Those servers cooperate in the context of **Subsystems** (e.g., UNIX) to provide a coherent set of services and interfaces to their ''users'' (Figure 1).

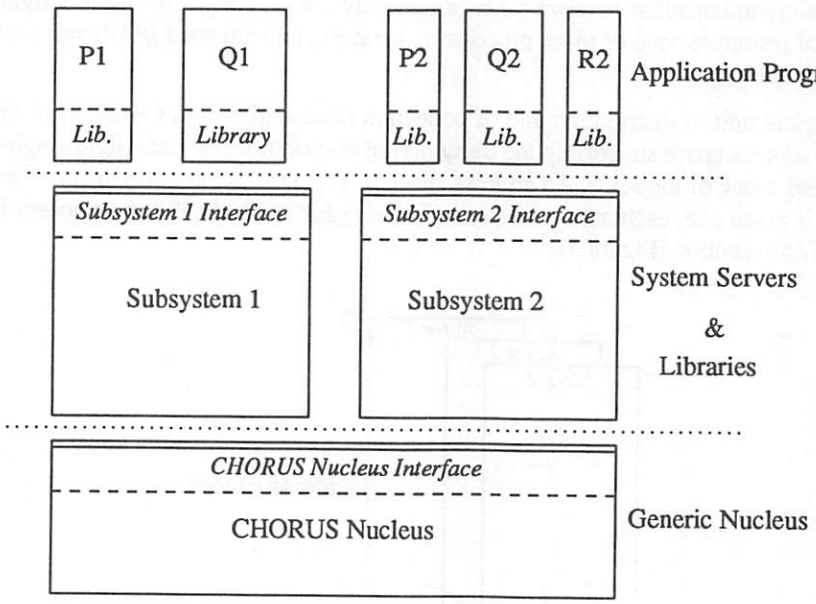


Figure 1. – The CHORUS Architecture

The CHORUS Nucleus (Figure 2) plays a double role:

1. Local services:
   It manages, at the lowest level, the local physical computing resources of a ''computer'', called a *site* by means of three clearly identified components:
   - allocation of local processor(s) is controlled by a *real-time multi-tasking executive*. This executive provides fine grain synchronization and priority-based preemptive scheduling,
   - local memory is managed by a *virtual memory manager*,
   - external events – interrupts, traps, exceptions – are dispatched by a *supervisor*.
2. Global services:
   An *IPC Manager* provides the communication service, delivering messages regardless of the location of their destination within a CHORUS distributed system. It may rely on external system servers (i.e., Network Managers) to operate all kinds of network protocols.

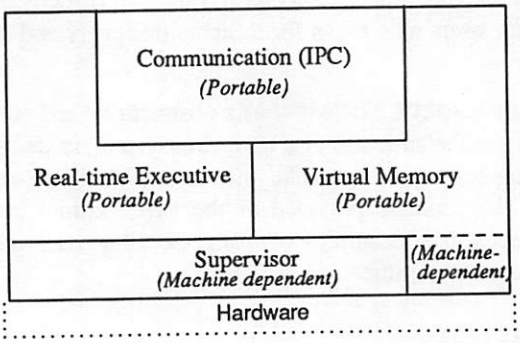


Figure 2. – The CHORUS Nucleus

2.2 The CHORUS Nucleus basic abstractions

The physical support for a CHORUS system is composed of a set of *sites* (''computers'', or ''boards''), interconnected by a communication *network* (i.e., a real network or a bus). A site is a tightly coupled grouping of physical resources: one or more processors, memory, and attached I/O devices. There is one CHORUS Nucleus per site.

The **actor** is the logical unit of distribution and of collection of resources in a CHORUS system. An actor defines a protected address space supporting the execution of one or more *threads* (lightweight processes) that share the address space of the actor. An address space is split into a *user address space* and a *system address space*. On a given site, each actor's system address space is identical and its access is restricted to privileged levels of execution (Figure 3).

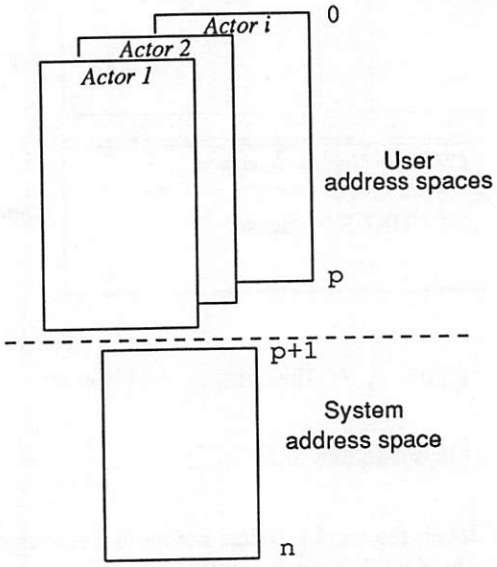


Figure 3. – Actor Address Spaces

Any given actor is tied to a site, and its threads are executed on that site. A given site may support many simultaneous actors. Since each has its own ''user'' address space, actors define protected *virtual machines*.

The **thread** is the unit of execution in a CHORUS system and is characterized by a *context* corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.). A thread is

always tied to one and only one actor. These threads share the resources of that actor and no other actor. Threads are scheduled by the Nucleus as independent entities. In particular, threads of an actor may run in parallel on the many processors of a *multiprocessor* site. The scheduling of threads is preemptive, based on their fixed priorities.

Besides the shared memory provided by the actor address space, CHORUS offers message-based facilities (referred to as *IPC*) which allow any thread to communicate and synchronize with any other thread, on any site. The CHORUS IPC permits threads to exchange messages either *asynchronously* or by *demand/response*, also called *Remote Procedure Call* (*RPC*). Its main characteristic is its transparency with respect to the location of threads: the communication interface is uniform, regardless of whether it is between threads in a single actor, between threads in different actors on the same site, or between threads in different actors on different sites.

A **message** is composed of a (optional) *message body* and a (optional) *message annex*. Both are untyped string of bytes. Message passing is tightly coupled with the virtual memory mechanism to enable data transmission without copy.

Messages are not addressed directly to threads, but to intermediate entities called *ports* (Figure 4).

A **port** is an address to which messages can be sent, and a queue holding the messages received but not yet consumed by the threads. A port can only be attached to a single actor at a time, but can be attached to different actors successively, effectively *migrating* the port from one actor to another.

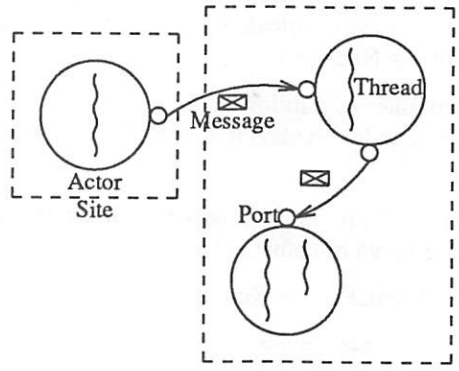


Figure 4. – CHORUS Nucleus basic abstractions

The notion of a port provides the basis for *dynamic reconfiguration*: this extra level of indirection between communicating threads, enables a given *service* to be supplied independently of a given actor. The servicing actor can be changed at any time, by changing the attachment of the port from the actor holding the initial thread to the actor holding the new one.

A **group** of ports connects those ports to a *multicast* facility: it allows one thread to communicate directly with an *entire group of threads* (via a group of ports); it provides also *"functional"* access to a service by selecting a server from a group of (equivalent) servers. A group is built by dynamically inserting ports into, and removing them from, the group.

Ports are globally designated with *Unique Identifiers* (UI's). A **UI** is unique in a CHORUS system. The CHORUS Nucleus implements a localization service, allowing threads to use these names without any knowledge of the location of the designated entities. UI's may be freely exchanged between actors.

Global names for other types of objects are based on UI's, but hold more information, such as protection information. These names are called *capabilities*. [Tane86] A **capability** is made of a UI and an additional structure, the *key*. When objects are Nucleus objects (e.g., actors), the UI is the global name for the object, and the key is only a protection key. When an object is managed by an external server (e.g., memory segments), the UI is the global name of a port of that server, and the semantics of the key are defined by the server. Generally, the key identifies the object within the server and holds the protection information. As with UI's, capabilities may be freely exchanged between actors.

2.3 Virtual Memory Management

The CHORUS memory management service[Abro89, Abro89a] provides separate address spaces (if the hardware gives adequate support), associated to actors, called **contexts**, and efficient and versatile mechanisms for data transfer between contexts, and between secondary storage and a context. The mechanisms are adapted to various needs, such as IPC, file read/write or mapping, memory sharing between contexts, and context duplication.

CHORUS memory management considers the data of a context to be a set of non-overlapping **regions**, which form the valid portions of the context.

Regions are mapped (generally) to secondary storage objects, called **segments**. Segments are managed outside of the Nucleus, by external servers called segment **mappers**. These manage the implementation of the segments, as well as the protection and naming of segments.

2.4 The Supervisor

The CHORUS Nucleus offers the following basic services to allow system actors to handle hardware events such as interrupts, traps and exceptions:

System threads may connect handlers (e.g., routines in the address space of their actor) to hardware interrupts. When an interrupt occurs, these handlers are executed. Several handlers may be simultaneously connected to the same interrupt, with control mechanisms to order or stop their invocation. After acknowledging the interrupt, handlers can communicate with other threads using asynchronous IPC or synchronization primitives provided by the Nucleus.

System actors may also connect routines to trap invocations, either as one routine or as an array of routines. In the latter case, the handler actually invoked is specified by a "service" number stored in a register of the machine.

Finally, an exception port or an exception routine can be associated with an actor, thus permitting Subsystem actors to deal with faults occurring within other actors.

TABLE 1. – Supervisor Interface

Supervisor interface	
`svConnect`	*Connect an interrupt or trap handler*
`svDisConnect`	*Disconnect an interrupt or trap handler*
`svCallConnect`	*Connect a trap handling table*
`svCallDisConnect`	*Disconnect a trap handling table*

3. The UNIX Sub-System

3.1 Overall structure

UNIX facilities may logically be partitioned into several classes of services according to the different types of resources managed: processes, files, devices, pipes, sockets. The design of the structure of the UNIX Subsystem in CHORUS puts emphasis on a clean definition of the interactions between these different classes of services in order to provide a true modular structure.

The UNIX Subsystem has been implemented as a set of System Servers, running on top of the CHORUS Nucleus. Each type of system resource (process, file, etc.) is isolated and managed by a dedicated system server. Interactions between these servers are based on the CHORUS IPC which enforces clean interface definitions (Figure 5).

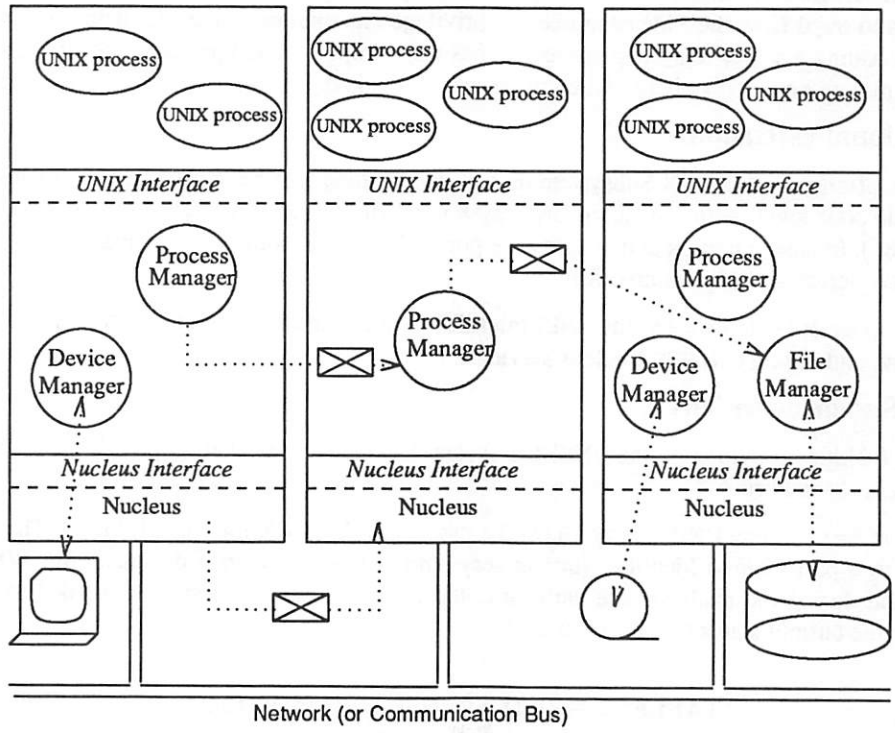


Figure 5. – UNIX as a Set of Independent Servers

Several types of servers may be distinguished within a typical UNIX Subsystem:

– **The Process Manager (PM)**
The Process Manager (PM) executes services directly related to UNIX process management (creation and destruction of processes, signals, etc.). The UNIX services have been extended to provide transparent access to distributed resources, so the PM's on the different sites of a network cooperate to provide remote services (such as remote `kill` or remote `exec`).
The PM manages the system context of each process. When the PM is not able to serve a UNIX system call by itself, it calls other servers as appropriate.

– **The File Manager (FM)**
The File Manager (FM) performs file management services. The current version is compatible with SYSTEM V.2 services and physical disk layout. New versions, compatible with SYSTEM V.3.2 and BSD 4.3 respectively are currently being integrated into CHORUS.
The FM also acts as a CHORUS *external mapper* for distributed virtual memory management by performing the `page_in/page_out` requests issued by the CHORUS Nucleus Virtual Memory Manager.

– **The Device Managers (DM)**
The Device Managers (DM) manage asynchronous lines, bit-map displays, *pseudo-ttys*, etc. and implement the UNIX *line disciplines*. Several DMs can run simultaneously on one site servicing different peripheral devices.

– **The Pipe Manager (PIM)**
A Pipe Manager implements UNIX pipe management and synchronization. `Open` requests for named pipes, received by File Managers are forwarded to a Pipe Manager. Pipe Managers may be active on every site, thus reducing network traffic when pipes are invoked on diskless stations.

– **The Socket Manager (SM)**
The Socket Manager implements BSD 4.3 socket services, providing access to TCP/IP protocols.

Those system servers can run either in *User space* or in *System space*. Those needing to connect some of their routines to *traps* (like the PM) or to execute privileged instructions (like I/O operations) run in system space. Loading a server in system space also has some impact on the performance of the server as it avoids extra memory context switches when the server is invoked.

3.2 Functional extensions

The interface offered by the UNIX Subsystem on a given machine, can be made binary compatible (i.e., at the executable code level) with a standard UNIX system taken as a reference (currently System V Release 3.2 on AT/386), to ensure complete user software portability. In addition, UNIX drivers can be integrated into a CHORUS Server with minimum effort.

CHORUS also provides extensions to the UNIX interface to take benefit of the distributed nature of the system and of the underlying CHORUS Nucleus services.

3.2.1 File System extensions

The naming facilities provided by the UNIX file system have been extended, to permit the designation of services accessed via Ports.

Symbolic Port Names (new UNIX file type) can be created in the UNIX file tree (Table 2). They associate a file name to a port Unique Identifier (this is very similar to UNIX device designation). When such a name is found during the analysis of a pathname, the corresponding request is forwarded to the port – marked with the current status of the analysis.

TABLE 2. – UNIX Symbolic Port System Calls

Symbolic Port System Calls	
`symport`	*create a symbolic port*
`readport`	*get the Unique Identifier associated with the symbolic port*
`lstat`	*do* `stat(2)` *on the symbolic port itself*
`unlink`	*unlink the symbolic port itself*

User written servers as well as system servers can be designated by such symbolic port names, thus allowing "users" to make extensions to the system dynamically (see Section 4.5). In particular, this is used to interconnect file systems and provide a global name space. For example, in Figure 6, ''pipo'' and ''piano'' are symbolic port names.

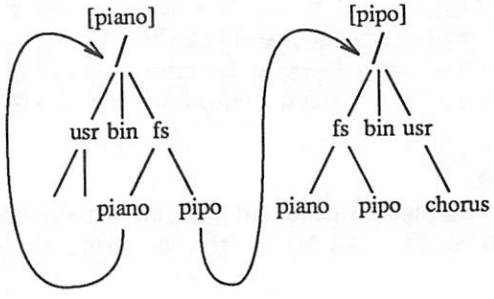


Figure 6. – Interconnection of File Trees

3.2.2 Process Management extensions

These extensions have been introduced to traditional UNIX services:

- The basic extension to process management is to enable remote creation (`fork(2)`) or execution (`exec(2)`) of processes. A "creation site" information has been added to the system context of UNIX processes. This information is inherited through `fork(2)` and `exec(2)`. It may be set by means of a new system call: `csite (SiteId)`. This information is used when a process is forking or exec'ing: on a `fork(2)`, the child process will be created on the site specified by `SiteId`. On `exec(2)`, the process will start the execution of the new program on the site specified by `SiteId`.

  Creating or moving processes on any site, implies that process identifiers are unique over the distributed system. Process identifiers used by UNIX servers are 32 bits long. UNIX processes can manipulate either PIDs of 16 bits (for binary compatibility reasons) or 32 bits which will allow them to address signals to remote processes. The new `pcntl (LONGPID)` system call sets a system context flag which enables a process to manipulate PIDs 32 bits long.

- At `exec(2)` time, processes can be dynamically loaded into the system space, provided that the text and data regions that they need are free. Such a process will execute at a privileged level, thus being able to execute I/O instructions.

- Processes can lower or raise their priority, thus allowing real-time applications to run on the UNIX Subsystem (see 4.4).

3.2.3 Other extensions

It is natural to provide UNIX processes with access to some of the services offered by the CHORUS Nucleus i.e., IPC, Virtual Memory and Threads. Such access is not provided by directly invoking the Nucleus but rather through the UNIX Process Manager, in order to eliminate inconsistencies. For example, if a UNIX process could create a thread by directly invoking the CHORUS Nucleus without the Process Manager knowing about it, this thread would not be able to issue UNIX system calls correctly.

Therefore, some CHORUS Nucleus services are not available at the UNIX Subsystem interface (e.g., no actor creation or deletion primitives), and some restrictions and controls are performed: e.g., forbid the creation of threads inside other UNIX processes and the use of the UNIX process identifier instead of the CHORUS actor Unique Identifier in calls such as `portMigrate`.

To clearly distinguish between the two levels of interfaces, UNIX primitives allowing access to CHORUS services have been prefixed by "`u_`" (e.g., `u_portCreate` instead of `portCreate`).

3.2.3.1 Virtual Memory services

UNIX processes can use the Virtual Memory services of the CHORUS Nucleus to create regions, map segments within a region, share regions, etc. They can thus gain access to the physical memory (e.g., for mapping bitmap memory).

3.2.3.2 Inter Process Communication

UNIX processes can create ports, insert ports into groups, and send and receive messages. They can migrate ports from one process to another. CHORUS IPC mechanisms allow them to communicate transparently over the network. Applications can therefore be tested on a single machine, and then distributed throughout the network, without any modification necessary to adapt to a new configuration. Using port migration or group facilities provides a sound basis for doing dynamic reconfiguration and developing fault-tolerant applications.

3.2.3.3 Multi-threaded UNIX Processes

Multiprogramming within a UNIX process is possible with **u_threads**. A u_thread can be considered as a lightweight process within a standard UNIX process. It shares all the process resources and in particular its virtual address space and open files. Each u-thread represents a different locus of control.

When a process is created by `fork(2)`, it starts running with a unique u_thread; the same situation occurs after `exec(2)`; when a process terminates by `exit(2)`, all u_threads of that process terminate with it.

A set of signal handlers is associated with each u_thread. Signal sent on an exception are delivered to the faulty u_thread (only); alarm signals are delivered to the u_thread which set the alarm; all other signals are broadcasted to all u_threads of the process. Signal handlers are executed on the stack of the u_thread

which set the signal handler. Thus full consistency with existing signal handlers is maintained. U_threads may issue UNIX system calls. For reasons of simplicity (i.e., efficiently insuring consistency of the process system context), these are serialized except blocking system calls such as `read(2)`, `write(2)`, `pause(2)`, `wait(2)`, `u_ipcReceive(2)` and `u_ipcCall(2)` (i.e., those interruptible by signals).

3.3 Implementation

3.3.1 Structure of a UNIX Process

A UNIX process can be viewed as one thread of control executing within one address space. Therefore each UNIX process is implemented as one CHORUS actor. Its UNIX system context is managed by the Process Manager. The actor address space is divided into memory regions for text, data and execution stacks.

In addition, the Process Manager attaches one *control port* and one *control thread* to each actor implementing a UNIX process. The control port and the control thread are not visible to the user of that process.

Control threads executing within process contexts share the process address space and can easily access and modify the core image of the process (e.g., stack manipulations on the reception of a signal, text and data access during debugging). They are also ready to handle asynchronous events received by the process (mainly signals). These events are implemented as CHORUS messages received on the control port (Figure 7).

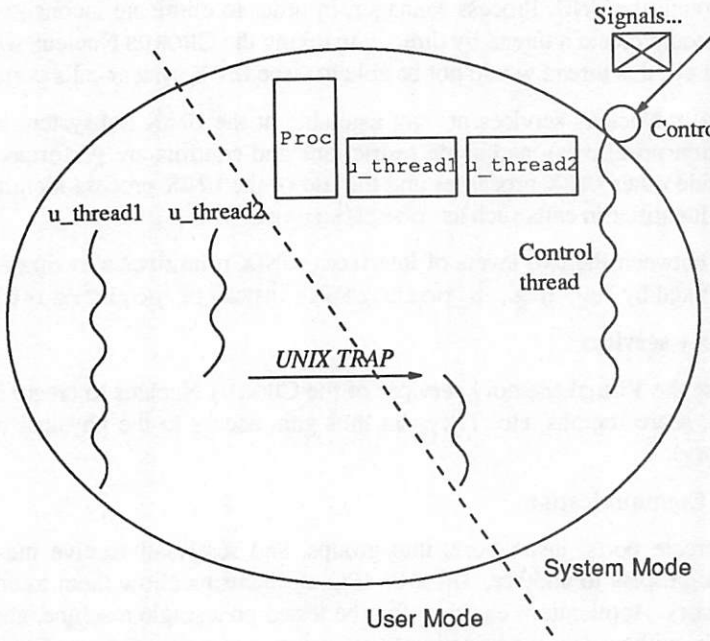


Figure 7. – UNIX Process as a CHORUS Actor

Because a process can be multi-threaded, the UNIX system context attached to one process has been split into two system contexts: one process context (`Proc`) and one u_thread context (`u_thread`).

Most services implemented outside of Process Managers are file related services. However, the file context of a Process (e.g., current and root directories, open files, `umask` and `ulimit` informations) are kept in the `Proc` structure, held by the Process Manager. This implies a specific protocol between PM's and other servers, as shortly outlined in Section 3.3.4.

Both system contexts `Proc` and `u_thread` are maintained by the Process Manager of the current process execution site. These contexts are accessed neither by the CHORUS Nucleus nor by other system servers. On the other hand the UNIX Subsystem is unable to see the internal Nucleus structures associated with actors and threads, the only way to access them is through Nucleus system calls (this is essential for allowing multiple Subsystems to co-reside on top of the same CHORUS Nucleus).

3.3.2 Process environment known by its set of ports

The semantics associated with ports by the CHORUS Nucleus – unique and global naming, addressing by IPC with location transparency – makes them extremely useful for designating system entities. The main advantages of using ports are the indirection that they provide between the process and its environment, and the robustness against the evolution of configurations. Port names stored in the process context are always valid whether the process itself migrates to another site (i.e., `exec` to a remote site) or if some of the entities to which they are related to migrate.

Used directly or embedded within capabilities, ports constitute the main part of a process environment. Embedded in capabilities, ports are used to designate process resources: e.g., open files or segments mapped into the process address space (`text`, `data`). But ports are also used directly to address processes.

Resources and capabilities

Every resource (managed by a Server) used by a process is designated internally by a capability: open file, open pipe, open device, current and root directories, text and data segments, etc. Such capabilities may be used to create regions in virtual memory; thus their structure is the one exported by the CHORUS Nucleus.

For example, opening a file associates the capability sent back by the appropriate server to the correct file descriptor. The capability is built with the port of the server that manages that file and the reference of the open file within the server. All requests on that open file (e.g., `seek(2)`, `close(2)`) are translated directly into a message and sent directly to the appropriate server.

Because the server of a resource is designated by a port, and because the localization of a port is transparent as part of the CHORUS IPC, the UNIX Subsystem does not have to locate UNIX servers.

Capabilities are computed and sent back by the servers. A server can thus delegate a service to another server, without clients knowing which actual server serves its requests.

3.3.3 The Process Manager

All of the UNIX Subsystem code concerning process management, signal handling, and the interface for the system calls accessible from a process, are in a single actor: the Process Manager (PM).

This actor is loaded when booting the system. Its code and data areas are initialized in the system memory space. The presence of a PM inside the system area makes it possible to implement system calls by using `traps`, as in UNIX (u_threads are running PM code after each system call), thus allowing binary compatibility with other UNIX systems.

The PM actor has the following resources:

- A *port* for receiving RPC requests addressed to the PM (remote `kill` and `exec`). This port is inserted in the *static group* of PM ports, used for locating a process.
- A *thread* dedicated to processing requests received on that port.
- A *thread* used for managing alarms. That thread is woken up each time an alarm arises and it sends a message to the control port of the actor that owns the u_thread which set the alarm.
- The data area of the actor. It includes, in particular, the Proc and the u_thread structures.
- A scratch area used to send and receive messages or to access stack areas of user processes (e.g., for mapping/demapping operations).
- The code area of the actor. Located in the system memory space, it is shared by all processes and executed by PM threads when processing remote requests and handling alarms, by u_threads when doing a system call, or by control threads (one per process) upon receipt of asynchronous messages addressed to processes (e.g., signals, children, death).

3.3.4 The File Manager

The FM is a system actor that has two ports on which it receives request messages. One of these ports deals exclusively with messages for paging virtual memory. The other port receives all other requests:

UNIX services, cooperation messages between PM and the FM, between Device Managers (DM) and the FM, and between the FM and other FMs in a distributed system. This port is called the "UNIX port"of the FM.

Once the initialization phase is over, several threads execute inside the FM. These threads process messages from one or the other of the two ports (Figure 8).

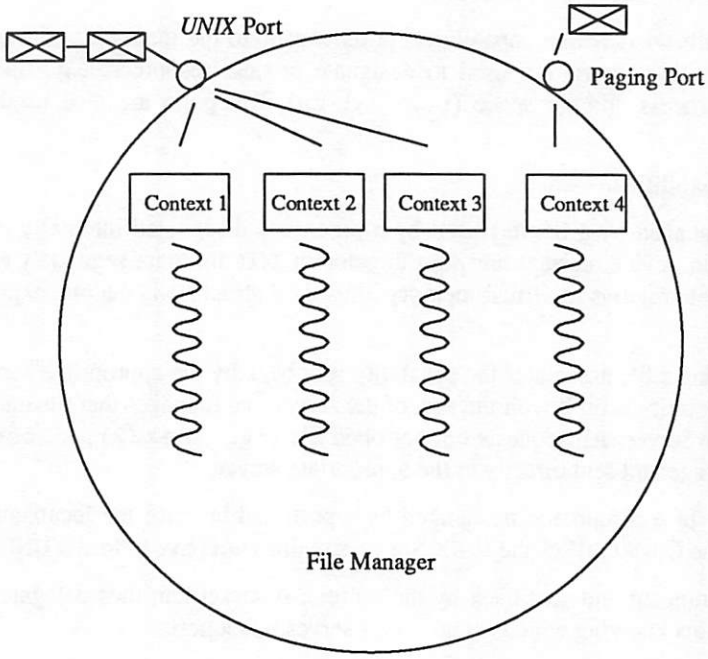


Figure 8. – File Manager Dynamic Structure

The general execution scheme for these threads is:

1. wait for a message to process,
2. initialize the thread context from the contents of the message,
3. invoke the required service,
4. prepare the reply message, and send it to the original requester,
5. return to (1).

The FM contains state information similar to that of a traditional UNIX file system. It owns and manages the following structures:

- A table of open files (one entry per `open(2)` performed), that contains in particular the flags used when opening the file (read and/or write access, etc.) and the current position in the file.
- A table of *inodes*, containing the memory images of the disk descriptors for the files currently in use.
- A table of the mounted volumes containing the volume descriptors of the disks currently mounted.
- A cache of the disk blocks, allowing the FM to minimize the number of physical disk accesses.

In addition, each thread that processes requests has an associated process context structure that simulates the system context that would be present in a traditional UNIX kernel (i.e., the U area). This context contains, for example, the identification of the user on whose behalf the thread is performing the request, the parameters of the request, and the global scratch variables equivalent to those of a UNIX kernel file system.

Because the context of the process on whose behalf the FM is processing the request, is not directly accessible to the FM, context information needed to serve the requests are included into the request messages, together with the system call parameters. The server includes in the replies those information necessary to update the file context of the process. Such a scheme is illustrated in Figure 8. Other servers such as Pipe, Device, or Socket Managers operate under a similar scheme.

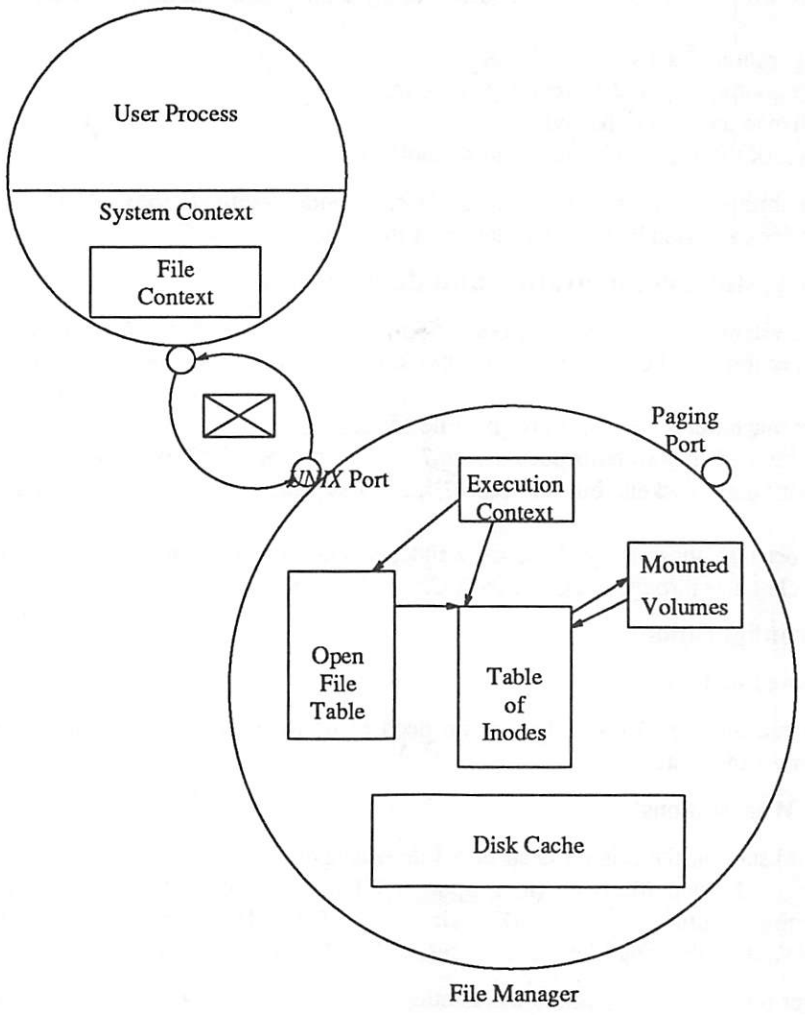


Figure 9. – Process and File Manager File Context

4. UNIX brought back to its original virtues

4.1 A tool-kit system

The structure of the UNIX Subsystem of CHORUS brings back to UNIX some of its original characteristics which have been gradually worn away by the thousands of hacker×years spent introducing new features into a monolithic kernel. The same ideas that UNIX had been promoting regarding the development of software tools have been applied by CHORUS to the operating system itself. They can be summarized as:

- make system servers implement only one type of service very simply and efficiently, rather than a lot of complicated features inefficiently,
- adapt existing servers rather than redoing everything from scratch, and fill the gaps by developing only those servers which are missing, when you want to build a new operating system (or extend an existing one).

Some of the servers in the UNIX Subsystem have been written from scratch (e.g., Process Manager, Socket Manager), while others have been adapted from existing UNIX kernel code (e.g., File Manager, Device Manager). In both cases the interdependencies and functions of each of the servers have been carefully designed, so that they can be combined in various ways to adapt the behavior of the resulting system to its user's needs. The servers have been made as flexible as possible so that they can be dynamically configured.

Some of the areas which most benefit from such a design and which will be examined in the following sections are:
- the static configuration of a distributed system,
- the dynamic (re)configuration of a distributed system,
- the ability to change the system behavior,
- the configuration of the system interface and semantics.

Most of these capabilities come from the basic services provided by the CHORUS Nucleus, but also from the way these services are used by the upper layers of the system.

4.2 Modularity, static configuration and distribution

As the UNIX subsystem is composed of a collection of servers, it is straightforward to adapt it to the hardware configuration of the system, or to the needs of the applications which run on such a configuration:
"No disk on your machine?"... "Don't take the File Manager"...
"No Terminal connected to your embedded system?"... "Don't load the Device Manager"...
"Your application uses sockets but no pipes?"... "Take the Socket Manager, but not the Pipe Manager"...
This is possible because these servers are truly independent from one another and because their only interface to their clients is through the CHORUS IPC.

4.2.1 Typical configurations

4.2.1.1 Standalone Machines

On a standalone machine, obviously, there is no need of network protocols, so the Network Manager need not to be part of the system.

4.2.1.2 Diskless Workstations

On a diskless workstation, there is no need of a File Manager. Only a Nucleus, a Network Manager, a Process Manager, a Device Manager (to support a bitmap and pseudo-ttys) and possibly a Socket Manager are required to provide a full UNIX environment. UNIX file system calls are converted into IPC requests by the PM, thus allowing transparent access to File Managers running on remote disk servers.

The Pipe Manager resides on the diskless workstation, but this is not mandatory. If it is not there, another equivalent server in the distributed system will serve the pipe requests of that station. Loading it on the station itself only provides better response time for accessing pipes, because it avoids accessing the network.

4.2.1.3 Multi-computers

From the point of view of a distributed system like CHORUS, the structure of a multi-computer (e.g., a hypercube) is very similar to the structure of a network of servers and workstations. The same configuration choices can be made: loading drivers only on the nodes where they are useful, loading a Socket Manager on the nodes providing connections with the outside world, loading a File Manager where disks are located. Only a Nucleus, a Network Manager and a Process Manager need to be present on each node to make that node look like a full UNIX system to application programs (on nodes running only one process of the application, this can actually be reduced to a simpler run-time system).

In fact, the version of the Network Manager running on a node not connected to an external network need not implement all the network protocols but only those handling inter-node communication. Network Managers running on nodes that provide access to an external network, must provide both families of protocols. This system architecture is being used on the EuroWorkStation developed in the EWS Esprit II Project 2569.

4.2.1.4 Multi-processors

The CHORUS kernel can run on symmetric multi-processor machines, providing its actors with uniform and simultaneous access to the processors.

The modularity of the UNIX subsystem allows benefiting directly from this facility: because UNIX services are implemented as independent servers, no synchronization is required between these servers.

Thus, on a four processor machine, each processor could have a thread running from a different UNIX server (e.g., PM, FM, DM, SM). This provides a very simple multiprocessing of UNIX services without having to worry about adding synchronization schemes to those UNIX servers which have been directly derived from current UNIX kernel code (FM, DM).

Moreover, some of the new servers, such as the Process Manager, have already been structured so that they can be themselves multiprocessed. Thus, several processes can invoke simultaneously a UNIX system call. On a four processor machine, two processes can `fork(2)`, another can issue an `open(2)`, and still another one a `read(2)`, all in true parallelism. Synchronization on global tables of the Process Manager is done at a fine level of granularity.

Regarding multiprocessing, servers implemented from UNIX code have the same level of granularity as in current UNIX kernels. Finer levels of granularity will be introduced in further releases of the system, using the synchronization primitives offered by the CHORUS Nucleus.

4.2.1.5 Embedded systems

For real time applications running in embedded systems, there may be no need or use for services other than those offered by the CHORUS Nucleus, the Process Manager and the Socket Manager. These will provide such applications with access to process and thread primitives, IPC, memory management, and connection to the hardware. Communication with the outside world can be done through the Socket Manager, using the services of the Network Manager.

These embedded applications may run in an environment without any other machine managed by CHORUS. In order to offer more flexibility (file access) and dynamism (loading/unloading programs, remote debugging) to such embedded applications, file services (including pseudo tty's), can be provided through a very simple File Manager which maps all CHORUS IPC file requests to socket communication. The requests carried through a socket connection are then processed by a UNIX process acting as remote server and running on any UNIX system.

4.2.2 Examples

The adaptability of the UNIX subsystem is clearly illustrated by the three following real cases.

4.2.2.1 Fault tolerant documents database server

A document database server that runs on a Motorola 68030 board plugged into a MacII[3] running CHORUS has been developed by an independent company. The database application runs on the board which also supports the disks (mirrored disks and/or an optical jukebox). The only services needed by the application are disk management and file access. Access to the database services is provided to the outside world (i.e., the MacII and other clients connected to the MacII by a network) via common memory shared by the MacII and the 68030 board.

The underlying system is composed of a CHORUS Nucleus (without the Network Manager) and a File Manager. A library was developed (from the Process Manager code, in one month) to transform every UNIX file system call into a CHORUS IPC request. The UNIX-like file context is thus managed in user's space in the same way as I/O streams are managed in the standard C library. Avoiding a full Process Manager saves memory space and provides better response time on file access by avoiding traps.

4.2.2.2 X terminal

CHORUS is being used in an X terminal product built by an independent company. The only program that runs in such a configuration is the X–server, which serves X–window requests coming from clients on other machines. It is to run in environments without other CHORUS machines.

The X terminal system is composed of a CHORUS Nucleus, a Network Manager, a Process Manager, a Socket Manager and a Device Manager (for keyboard and mouse drivers). There is no need for a Pipe or File Manager. Because the X–server still does some `open(2)` calls at init, to access devices, a small

3. MacII is a trademark of Apple Computers

dedicated File Manager has been developed (in less than a week) to serve these specific requests.

4.2.2.3 EuroWorkStation

The EuroWorkStation being developed in the EWS project previously mentioned is a high level scientific workstation. It is organized around a SPARC based symmetric multi-processor (as the main board(s)) which uses a Multibus II. An Intel 80386 based coprocessor board can be added on the MultiBus II. Access to Ethernet, FDDI and SCSI bus are also provided. Bitmap display is done (remotely) to an X–terminal. Planned coprocessors include 3D graphics, Lisp and a simulation engine.

The CHORUS system will run on such a configuration with a full UNIX Subsystem on the main board(s). On the co-processors, only a CHORUS Nucleus, a Network Manager (adapted to the Multibus II) and a UNIX Process Manager will be loaded. This will allow users to dynamically load programs on the co-processor from the basic workstation. To access the specific hardware of the coprocessors, an adapted version of the Device Manager will be loaded on each co-processor.

4.3 Dynamic Configuration

4.3.1 Sub-system configuration

The only UNIX server that needs to be loaded at boot time is the Process Manager, (and the File Manager if there are disks connected to the machine). With the CHORUS Nucleus it provides enough services to dynamically load other servers when needed. A very simple access to the system console (if any) is provided by the CHORUS Nucleus. Of course, UNIX-like tty line disciplines are not implemented in that case. This allows a shell to run on the console or to provide input/output on a terminal for processes which do not need UNIX terminal management.

The dynamic loading of UNIX servers is achieved through the standard UNIX interface: `fork(2)` and `exec(2)`. The Device Manager, the Pipe or the Socket Managers can be loaded by `init(1)` from the UNIX System V "*/etc/inittab*" file. This results from two services offered by the Process Manager to the super-user:

- the ability to load a UNIX process into the system address space, as specified at link time. Of course, if the virtual addresses needed by the process to be loaded are already used, `exec(2)` will fail.
- the ability to dynamically connect user specified routines to hardware interrupts. Such routines are invoked each time the interrupt occurs, until the routine has been disconnected from that interrupt.

Each device driver may be implemented in a separate Device Manager (e.g., for bitmaps, RS232 interfaces, tapes), these drivers can be loaded only when they are needed and can then be unloaded when they become useless. These Device Managers and the Pipe and the Socket Managers as well, are in fact UNIX processes and thus can take advantage of the UNIX services offered by the Process Manager and the File Manager, allowing them, for example, to record events into log files using standard UNIX system calls.

This dynamism can also be used to change the UNIX behavior of the subsystem. This was used in the Aphrodite Esprit Project 1535[Mino88] to build a host/target development, remote execution and debugging environment for real-time applications. A simple window manager was developed on top of the UNIX subsystem. For reasons of efficiency, this Window Manager (loaded as a UNIX process) catches interrupts directly from the mouse and the keyboard of an AT/386 computer. When the Window Manager is running, it diverts every interrupt from the Device Manager; the UNIX shell is thus blocked waiting for input. When the Window Manager becomes useless, quitting it disconnects the interrupt from the Window Manager routine and the Device Manager continues to work unaffected.

4.3.2 Server configuration

Since UNIX services are implemented by servers built on top of the CHORUS Nucleus interface, it is very easy to dynamically adapt the resources of each server to the actual needs of the application. This dynamic configuration relies on the following:

4.3.2.1 Adding threads in servers

When a process is created (or servers loaded at boot time), it runs only one thread. Other threads may be created dynamically. For example, the File Manager and the Device Manager create additional threads during their initialization phase. As servers are accessed through IPC or traps, their respective number of

threads can be adapted to their varying needs without stopping the system (to reload newly configured servers). Of course, servers may also delete useless threads.

In particular, such a scheme is used by the File Manager when a diskless site is coming up. The File Manager is told to create new threads in order to serve such a site. When the site shuts down, those threads become useless and are deleted.

Currently, these configuration parameters are monitored by the system administrator using dedicated commands. Another approach would be to let servers create a new pool of threads themselves when the number of idle threads goes under a low water mark and to delete them when this number raises over a high water mark. Such configuration issues are discussed in Section 5.

4.3.2.2 Space management in servers

Adding threads in UNIX servers is not the only configuration issue. One must also configure the memory resources to the size appropriate to the use of the system. For example, if one wants to raise the maximum number of processes that may run simultaneously on a processor, the Process Manager must resize its Proc table. Dynamically resizing tables requires allocating memory space for new entries, and algorithms for allocating, freeing and searching entries that do not depend upon the physical organization of the tables. Space allocation within an actor can be done by invoking `rgnAllocate` with the size needed. This call returns the address of the newly allocated memory region.

The second problem is solved (or eased) by the use of C++. Basic tools for managing "*pools*" of elements have been developed. The process table is such a pool. When creating a new process, the Process Manager invokes `"PROC.allocate"` to get a free entry. When the process exits, it calls `"PROC.free"` to free the entry in the table. The implementation of the pool is hidden by these functions. The current implementation relies on linked list mechanisms. This pool mechanism is used by the servers for every new table that has been introduced. It is not used for tables allocated and managed by C code coming from an existing UNIX kernel implementation. This will be done in a future release of the system.

Having servers implemented as actors and allocating their internal tables in virtual regions eases the management of the usage of physical memory. Servers such as the Socket Manager, the Pipe Manager and even the Process Manager may be paged out without disturbing the service. Servers that connect code to interrupt are locked in memory to avoid page faults upon reception of an interrupt.

4.4 Real-Time operation

Making use of the real-time scheduling provided by the CHORUS Nucleus facilitates development (static or dynamic) of different scheduling policies in the Subsystem. The UNIX Subsystem of CHORUS takes advantage of this facility to provide a real-time execution environment.

4.4.1 Changing the priorities of a server

The CHORUS Nucleus schedules threads on a fixed priority basis. Priorities range from 1 (the highest) to 255 (the lowest). Threads running with a priority between 128 and 255 are time-sliced at the same priority level. In a usual CHORUS configuration, UNIX processes run at priority 128 and UNIX servers at priorities 64 to 68. As explained earlier, this gives the user the ability to run real-time processes with a higher priority than standard UNIX processes; they may even run with a priority higher than the UNIX servers!

If the range of priorities used by the UNIX servers is not adequate for a given system, the priorities of the UNIX servers can be changed either statically by recompiling the servers, or dynamically by sending them a request to change their priority through the `threadPriority` system call. In both cases, priorities of these servers may be lowered or raised as needed. In any case, attention should be given to the consistency of the new set of priorities used by the servers: it may not be very meaningful to have UNIX servers running at the lowest priority, while standard UNIX programs are running at the highest one...

4.4.2 Delaying interrupt processing

UNIX servers which deal with hardware interrupts (File Manager, Device Manager) execute interrupt code at interrupt level, just as in a standard UNIX kernel, providing equivalent response times to those provided by such UNIX kernels.

However, the File Manager and the Device Manager may also run in another mode. Upon reception of an interrupt they can just post an event to a dedicated thread (named the "*Interrupt Thread*") created at init time, and then return, after the interrupt level has been acknowledged. Posting an event can be done by means of synchronization primitives offered by the CHORUS Nucleus. The real interrupt routine of the driver will be executed when the Interrupt Thread gets the processor, depending on its priority. This "delayed processing of interrupts" minimizes the time during which interrupts are masked, leaving more time for real-time processes to run and deal with their own interrupts. In this mode, critical sections management inside the File Manager (or Device Manager) is done without actually masking the interrupts. Of course, this mechanism requires additional scheduling for each interrupt which processing has been delayed. This implies that the UNIX response time will be affected, therefore this is only useful when executing real-time applications.

The system administrator can dynamically change interrupt processing from immediate processing (as in standard UNIX implementations) to delayed processing mode, and vice versa. In addition, the priority of the thread managing the interrupts can be fixed dynamically. This feature has been implemented in the File Manager as well as in the Device Manager. The processing mode of interrupts can also be chosen or even be frozen when compiling the server.

This functionality allows the development of real-time applications in a standard UNIX environment while editing or compiling the application. Once the application has been written, prior to executing or testing it, the priorities of the UNIX servers can be lowered, thus minimizing masking periods to enable the application to react correctly. When the application is finished running, the priorities of the UNIX servers can be reset to their previous values making interrupts immediately processed and recovering the initial system behavior. Such facilities avoid the necessity of stopping and reloading the system(s) (the UNIX development system and the real-time execution system). This mechanism is roughly analogous to that provided by the UNIX shell with job control, which allows one to stop processes and then to restart them later. Here the system is not stopped but only "niced".

4.5 Extending system services

Another aspect of the flexibility provided by the CHORUS implementation of UNIX is the ability to dynamically tailor the services offered by the system to the user's needs.

4.5.1 Adding system calls

As illustrated earlier, the UNIX Subsystem as been extended in two ways: by a minimum set of extensions to standard UNIX interfaces for distributed environments, and by CHORUS extensions to provide UNIX processes with the IPC, threads and virtual memory services offered by the CHORUS Nucleus. In fact, the second category of extensions may be made accessible or not to UNIX processes.

The Process Manager uses `svCallConnect` to connect routines to traps (i.e., system calls). But UNIX services and CHORUS specific services are not implemented through the same "*sysent*" table, in order to facilitate the adaptation of the Process Manager to provide binary compatibility with a given UNIX implementation on a given hardware.

In addition to the standard UNIX interface, the Process Manager provides a service which permits extensions to be made available or unavailable, thus tailoring the interface to particular needs. This capability will be completed with dynamic loading of pieces of the Process Manager code, when compilers generating position independent code are more widely available.

This connection of routines to traps can also be used by UNIX processes loaded in system space to extend the current interface with new services accessible through traps. This allows extensions to the UNIX system to be written or provides a new system interface (e.g., an Object Oriented System) on top of the CHORUS Nucleus simultaneously with the UNIX interface.

4.5.2 Enriching the UNIX semantics

Another way to provide more services is to use the file system's symbolic port mechanism. This has been used in a research project[Coyo89] that transparently provides duplicated files to UNIX. It does so without modifying either the interface, existing programs, or even the Process Manager or the File Manager.

At init time, the Duplicating Server creates a port and records it in the UNIX file system. Each time the File Manager encounters this port when analyzing a pathname in one of its requests, it redirects that request to that port. The Duplicating Server can then examine the request, duplicate it and send two resulting requests to two File Managers . After receiving the two responses, it replies to the client process, as if it was the initial File Manager.

Example:

If the sub-tree of files to be duplicated starts at the directory "/users/fa/srcdir", the server creates the symbolic port "/users/fa/srcdir.dup". To create a duplicated file (say a source file), invoke your favorite editor with the following pathname "/users/fa/srcdir.dup/hello.c". When writing, the file is updated on two repository file systems, as set by configuration parameters. Afterwards, invoke `make` and run your program; even if one of the repository file system fails during the make, the compiler will finish correctly!

This mechanism of request redirection is in fact very similar to I/O redirections or pipes in UNIX. In the above case, the only thing to be careful of, is to respect the protocol between the Process Manager and the File Manager. This protocol is actually part of the UNIX subsystem interface, and thus can be quite easily used.

4.5.3 Static Extensions

There is, in the system description of a process, an "*Extend*" class whose member functions are invoked on process system calls such as `fork(2)`, `exec(2)`, `exit(2)`, allowing system writers to add functionality to UNIX processes, by pure extension of the CHORUS code. This hook has been used to easily implement a CASE/PCTE UNIX Subsystem (on AT/386) on top of the CHORUS Nucleus.

4.6 Examples

4.6.1 Development of the Pipe Manager

In the early versions of the UNIX Subsystem, pipes were implemented within the File Manager (derived from System V code). Since then, pipe management has been extracted from the File Manager and rewritten as a UNIX process which can invoke any UNIX system services.

At init time, a Pipe Manager creates a port and inserts it into a group representing the pipe service. It then enters an infinite loop: waiting for incoming messages carrying requests (e.g., pipe creation, read, write), serving the request, replying to the request. As it uses only IPC to receive and reply to requests, it can be invoked and tested by user programs using the IPC interface of the UNIX Subsystem. This also allows running and testing this new implementation without disturbing the service provided by the running UNIX Subsystem, as pipes services are still provided by the File Manager.

Pipe management does not deal with either traps or interrupts, so the Pipe Manager does not need to be loaded in the system address space. It gets the address space protection of any UNIX process, which makes it easy to debug. Other benefits from having a system server be a UNIX process are that traces may be redirected to a file (using shell mechanisms), a crash of the server does not affect the system as a whole, and standard UNIX debuggers such as `sdb` can be used.

However, once fully tested, the Pipe Manager is relinked and loaded into the system address space, thus avoiding additional memory context switches when it is invoked. Finally, the pipe routine of the Process Manager needs to be modified to invoke the new Pipe Manager instead of the File Manager in case of pipe system calls. Only then does the system needs to be stopped and reloaded with the new version of the Process Manager.

4.6.2 Development of a new file system

Developing a new version of the File Manager follows the same steps than those outlined above for the Pipe Manager, except that disk drivers perform privileged instructions for I/O operations, and symbolic ports can be used to connect the file tree managed by the new File Manager under test to the file tree managed by the current operational File Manager.

To access disks from user space, the following mechanism has been used. To be allowed to access privileged instructions a thread must be executed in the system address space. Before loading the File Server being tested, a small process that connects two functions to traps using `svCallConnect` and

one function to the disk interrupt using `svConnect`, is loaded into the system address space. One of the trap calls is used to perform privileged I/O instructions to start the I/O. When the driver needs to issue such instructions, it just does the corresponding trap. The other trap call is used to wait for an incoming interrupt, it is used by the thread dedicated to delayed interrupt processing. When this thread starts, it enters an infinite loop: wait for interrupt with the trap function, and trigger the appropriate interrupt routine. When an interrupt occurs, the connected function is activated by the CHORUS Nucleus. This function posts an event which is awaited by the trap function called by the Interrupt Thread.

When the new File Server has been initialized, it creates a symbolic port in the system file tree, say "/tmp/newfs". Each access to files such as "/tmp/newfs/users/fa/myfile" will thus be received and served by the new File Server as an access to the file "/users/fa/myfile". This allows the full testing of the new File Server using standard UNIX utilities.

Of course, the new File Manager needs to be tested either on a machine with two disks or on a machine with one disk, booted as a diskless station, using a remote file system.

To replace the current version of the File Manager with the new one, the system must be stopped and reloaded. Avoiding stopping the system would imply that File Managers are stateless, or that they can transmit their current state to each other, which is somewhat complex to implement.

5. Lessons and open issues

The UNIX Subsystem on CHORUS shows clearly all the benefits one can gain from modularity in operating system development. However, improvements can still be made and alternative solutions to some of the issues raised by such an implementation are worth considering. Some of these are currently being studied in new versions of the system.

5.1 Performances

Regarding performances, modularity is not as expensive as is usually thought. Table 3 summarizes some initial performance measurements done on a COMPAQ 386/25. It compares the UNIX Subsystem of CHORUS with the Microport system.

TABLE 3. – Performance of the UNIX Subsystem

Primitives	CHORUS	Microport
`getuid`	*85 µs*	*80 µs*
`sbrk(0)`	*95 µs*	*128 µs*
`read (1Kb)`	*146.8 Kb/s*	*107.2 Kb/s*
`write (1Kb)`	*121.2 Kb/s*	*56.6 Kb/s*
`pipe (4096)`	*415 Kb/s*	*1212 Kb/s*
`exec`	*17 ms*	*37 ms*
`fork`	*17 ms*	*26.5 ms*

The read and write tests work on a 2 Megabyte file, 1 Kilobyte at a time. The pipe test writes and reads 4 Kilobytes blocks through a pipe. These measurements illustrate the viability of implementing a system as a set of servers without loss of performance. This topic, which is discussed in Section 5.3 leads to some other performance improvements, which will be illustrated there.

5.2 Modularity

Modularity has proven to be very convenient and powerful to adapt the system to hardware configurations. More modularity could be obtained in particular with the extraction of disk drivers from the File Manager to have them run in separate actors. This would allow File Managers to deal easily with remote disks, thus permitting to access floppy disks of diskless stations without any File Manager running on such a station.

5.3 Servers and Threads

As the UNIX service has been split into independent servers, more system resources are needed in order to be able to respond to client requests than in other UNIX kernel implementations. For example, for every u_thread created in a user process one should create one thread in the File Manager, one in the Device Manager, one in the Socket Manager, etc. Only such a policy can insure that system resources will be numerous enough to respond to client requests. This is especially true for servers in which threads can be blocked for a long or infinite time waiting for an incoming event: read on a terminal for Device Manager for example. While a server thread is blocked, other client requests cannot be served by that thread. If all the threads are blocked for reading, no process can write on terminals any longer! In other words, this means that as modularity increases, resource consumption rises, overloading the Nucleus tables with (most of the time) idle threads.

In fact, it is possible to configure the servers in such a way as to much diminish the problem, although without eliminating it. Mechanisms are being studied to transparently transform Remote Procedure Calls to local routine calls when the destination port is located on the same site than the sender. Thus a server is executed as a "monitor" by the u_threads which issued the system call. As a result, though modularity is preserved, the consumption of system resources is lowered. Threads running in servers only serve incoming remote requests.

Some preliminary developments in that direction have clearly shown its promise. A particular protocol between the Process Manager and the File Manager has been developed to simulate such a behavior, and to permit some real measurements. This transparent transformation of RPC into routine calls impacts the system in two other ways:

- When a process invokes a server, the code of the server runs at the priority of the process. High priority processes can be served with respect to their priority. When a real RPC is performed, the request is performed at a standard priority as defined in the server. Thus, this transformation makes it possible to provide users with a more real-time system.
- Executing server code in the context of the calling thread avoids context switches improving system performance. Some measurements have been done on the system emulating this feature. In this system, transformation of RPC into routine calls has been done for read and write operations. Results are shown in Table 4.

TABLE 4. – Performance of the UNIX Subsystem when converting RPC

Primitives	CHORUS with true RPC	RPC converted to routine call
`read (1Kb)`	*146.8 Kb/s*	*211.2 Kb/s*
`pipe (4096)`	*415 Kb/s*	*1240 Kb/s*

5.4 Caching

The distributed file system provided by the UNIX Subsystem is based on direct access of the client to the appropriate File Manager by means of the CHORUS IPC. This makes it possible to maintain full consistency with UNIX file system semantics.

An important drawback of such a choice is that there is no caching of remote data on the client side. Rather than having File Managers cooperate to cache remote data, the use of virtual memory mechanisms is being studied to implement file access. This still avoids loading a File Manager on a diskless station. Using virtual memory services makes it possible to take advantage of its caching mechanisms. Open files can be manipulated as segments cached by the virtual memory manager but not necessarily mapped into a particular region.

5.5 Symbolic Ports

Symbolic ports allow transparent interconnection of the UNIX name space, and provide a powerful extension mechanism. This is done without any lexical exception in pathnames. But introducing a new file type in the UNIX world implies that some standard utilities (less than 10) must be modified to take this new file type into account, e.g., `fsck(1)`, `cp(1)`, `find(1)`, `test(1)`. In fact, lexical exception is avoided by a semantic exception!

A general mechanism allowing to transparently (either from a lexical and a semantic point of view) connect servers to any node of a UNIX file tree seems more appropriate and convenient. Feasibility of such a mechanism is being investigated.

6. Conclusion

Making the CHORUS Nucleus *generic* prevented the introduction of "features" with "heavy" semantics. For example, features such as application-oriented protocols, fault tolerant strategies, do not appear in the CHORUS Nucleus. However, it provides the building blocks to construct these features inside subsystems.

On the other hand, CHORUS provides effective, high performance solutions to some of the issues known to cause difficult problems to system designers, mainly system (re)configuration (static and dynamic), adaptability, extensibility, and debugging, which is eased by isolating resources within actors and by communicating by means of messages providing explicit and clear interactions.

The CHORUS modular structure has been very successful, allowing to provide binary compatibility with UNIX, while keeping the implementation well structured, portable and efficient.

All these principles were those on which UNIX was initially designed 20 years ago on a standalone time-sharing computer. Networks and multi-processors introduce today new features and constraints that force one to "rethink" the internal structure of UNIX in order that it still be a modern operating system. CHORUS obviously shows that UNIX can be reminded of its original virtues and, while still keeping its standard interface for application programs portability, (r)evolve to the next generation of systems...

7. Acknowledgements

Vadim Abrossimov, Ivan Boule, Hugo Coyote, Corinne Delorme, Jean-Jacques Germond, Lori Grob, Marc Guillemont, Sylvain Langlois, Pierre Léonard, Pierre Lebée, Ian Liang, Jim Lipkis, Marc Maathuis, Denis Metral-Charvet, Will Neuhauser, Maria-Inès Ortega, Bruno Pillard, Didier Poirot, Eric Pouyoul, François Saint-Lu and Eric Valette contributed, each with a particular skill, to the CHORUS-V3 implementation on various machine architectures.

Many thanks are addressed to the referees for their helpful comments, and additional credits to Lori Grob for improving the readability of this paper.

8. References

[Abro89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro, "Generic Virtual Memory Management for Operating System Kernels," *Submitted for publication*, (April 1989), p. 20.

[Abro89a] Vadim Abrossimov, Marc Rozier, and Michel Gien, "Virtual Memory Management in Chorus," in *Lecture Notes in Computer Sciences, Workshop on Progress in Distributed Systems and Distributed Systems Management*, Springer-Verlag, Berlin, Germany, (18-19 April 1989), p. 20.

[Acce86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proc. of USENIX Summer'86 Conference*, Atlanta, GA, (9-13 June 1986), pp. 93-112.

[Alve88] Jose Alves-Marques, Roland Balter, Vinny Cahill, Paulo Guedes, Neville Harris, Chris Horn, Sacha Krakowiak, Andre Kramer, John Slattery, and Gérard Vandome, "Implementing the Comandos Architecture," in *Esprit'88: Putting the Technology to Use*, North-Holland Publishing Co., (November 1988), pp. 1140-1157.

[Cher88] David Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333.

[Herr88] Frédéric Herrmann, François Armand, Marc Rozier, Michel Gien, Vadim Abrossimov, Ivan Boule, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, "CHORUS, a New Technology for Building UNIX Systems," in *Proc. of EUUG Autumn'88 Conference*, EUUG, Cascais, Portugal, (3-7 October 1988), pp. 1-18.

[Li86] Kai Li and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems," in *Proc. of Principles of Distributed Computing (PODC) Symposium*, (1986), pp. 229-239.

[McJo88] Paul R. McJones and Garret F. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs," Technical Report 21, DEC Systems Research Center, Palo Alto, CA, (September 1988), p. 100.

[Mino88] Régis Minot, Pierre Courcoureux, Hubert Zimmermann, Jean-Jacques Germond, Paolo Alvari, Vincenzo Ambriola, and Ted Dowling, "The Spirit of Aphrodite," in *Esprit'88: Putting the Technology to Use*, North-Holland Publishing Co., (November 1988), pp. 519-539.

[Mull87] Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers 1984 -1987*, CWI Tract No. 41, Amsterdam, Netherlands, (1987), p. 309.

[Pres86] David L. Presotto, "The Eight Edition UNIX Connection Service," in *Proc. of EUUG Spring'86 Conference*, Florence, Italy, (21-24 April 1986), p. 10.

[Rash87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, (October 1987), pp. 31-39.

[Rozi88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Journal*, vol. 1, no. 4, The Usenix Association, (December 1988), pp. 305-370.

[Tane86] Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse, "Using Sparse Capabilities in a Distributed Operating System," in *Proc. of IEEE 6th. International Conference on Distributed Computing Systems*, CWI Tract No. 41, Cambridge, MA, (19-23 May 1986), pp. 558-563.

[Wein86] Peter J. Weinberger, "The Eight Edition Remote Filesystem," in *EUUG Spring'86 Conference*, Florence, Italy, (21-24 April 1986), p. 1.

[Coyo89] Hugo Coyote, "Spécification et Réalisation d'un Système de Fichiers Fiables pour le Système d'Exploitation Réparti CHORUS," PhD.Thesis, Université Paris VI, Paris, France, (June 1989), p. 300.

EXPERIENCE WITH THE DISTRIBUTED DATA STRUCTURE PARADIGM IN LINDA

M. Frans Kaashoek
*Henri E. Bal**
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Email: kaashoek@cs.vu.nl

ABSTRACT

Several parallel programming languages provide support for an unusual communication paradigm called distributed data structures, and a programming style called replicated worker [Bal et al. 1989b]. The best known and most widely used language that does so, is Linda, developed by D. Gelernter and colleagues at Yale University [Ahuja et al. 1986; Carriero et al. 1986; Carriero and Gelernter 1989; Gelernter 1985]. Because of this support, its designers claim that parallel programming in Linda is conceptually not harder than sequential programming.

We have implemented several non-trivial Linda programs using distributed data structures and the replicated worker style. This paper gives a second opinion on the suitability of these concepts. For certain classes of applications, serious problems reduce the amount of parallelism achieved using replicated workers. In addition, we argue that although the distributed data structure paradigm in Linda has a clean and elegant semantic interface, nonetheless Linda's support of distributed data structures is at too low a level.

1. INTRODUCTION

As parallel and distributed systems are becoming more commonplace, high-level languages for programming these systems are emerging. At the moment there are more than 100 of them [Bal et al. 1989b]. These languages can be divided into two broad categories according to their communication model: languages in which processes interact through shared

* This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

variables (e.g., Mesa, Edison, and Concurrent Pascal [Geschke et al. 1977; Brinch Hansen 1981; Brinch Hansen 1975]), and languages in which processes interact through message passing (e.g., CSP, Concurrent C, and SR [Hoare 1978; Gehani and Roome 1986; Andrews et al. 1988]). The first category is intended for tightly coupled systems, where at least part of the primary memory is shared. The second category is intended for loosely coupled systems, where processes have access to only their own local memory. They communicate by sending messages over a communication channel, such as a point-to-point link or a local-area network [Tanenbaum and van Renesse 1985].

In this paper we will study a communication paradigm that supports the shared variable paradigm on loosely coupled systems: distributed data structures. The best known language that supports this paradigm is Linda [Ahuja et al. 1986; Carriero et al. 1986; Carriero and Gelernter 1989; Gelernter 1985]. We will argue that the distributed data structure paradigm is a high-level concept, but that Linda provides a low-level implementation of this paradigm. We will do so by studying the distributed data structures solutions to two applications: a distributed backtracking package, and the all-pairs shortest paths problem.

In the following section we will briefly describe the paradigm itself. In Section 3 we describe Linda. In Section 4 we will show how the paradigm is used in solutions of the two applications and their implementation in Linda. In Section 5 we will evaluate the paradigm and Linda. In Section 6 we will present our conclusions.

2. DISTRIBUTED DATA PARADIGMS

A *distributed* data structure is a data structure that can be manipulated by multiple processes simultaneously through a set of operations [Carriero et al. 1986]. The operations on a distributed data structure must be indivisible. As an example, one can imagine a distributed set, with operations to add or to remove elements, test for set membership, and so on. The programmer can think of these operations as being executed in some sequential order. The implementation, however, can fully utilize the properties of the underlying computer architecture to allow maximum parallelism and to minimize overhead costs. The set can be put in shared memory, if that is available, and be protected by lock variables. The set can also be partitioned (split up) over several processors, for example, by letting one processor maintain the even elements and another one the odd elements. Alternatively, the set (or part of it) can be replicated on several processors. Operations such as testing for membership can then be performed on a local copy. The implementation can allow multiple read operations (like testing for membership) to execute simultaneously. It is even conceivable that multiple write operations (e.g., adding or removing elements) are to proceed in parallel.

The *Replicated worker* style is a programming style that differs substantially from more traditional styles for distributed programming, like the client-server. In the latter, work is usually split up among several communicating processes. In the replicated worker style [Carriero et al. 1986], however, workers are replicated rather than partitioned. There are p replicated processes, one for each processor. The work to do is stored in a distributed data structure accessible by all worker processes. Each process repeatedly takes some work from the data structure, performs it, puts back the results in the data structure, and possibly generates some more work. All workers perform essentially the same kind of task, until all work is done. The workers are loosely coupled; they only interact through the data structure. The replicated worker style is similar to the shared memory style for machines with shared memory.

Providing multiple processes with direct access to the same data structure is a major departure from message based languages. In such languages, data structures are typically encapsulated by a specific "manager" process that serializes all access to the data. The distributed data structure paradigm makes it possible to implement data structures in a safe way. The combination of replicated workers and distributed data structures has several advantages. In principle, any number of processes can be used (including just one). Extra processors are only used to create more workers. Usually, adding more processors means faster program execution. Also, the work is automatically and fairly distributed among the workers. Finally, process management is easy, as there usually is only one process per processor. In particular, many process switches are eliminated.

3. LINDA

Linda is based on the *Tuple Space* (TS). TS is conceptually a shared memory, although its implementation does not necessarily require physical shared memory. TS is a single global memory shared by all processes. The elements of TS, called tuples, are ordered sequences of values, similar to records in Pascal [Wirth 1971]. For example

```
["jones", 31, true]
```

is a tuple with three fields: a string, an integer, and a boolean. Three atomic operations are defined on TS: **out** adds a tuple to the TS, **read** reads a tuple contained in the TS, and **in** reads a tuple and also atomically deletes it from the TS. Unlike usual shared variables, tuples do not have addresses. Rather, tuples are addressed by contents. A tuple is denoted by specifying the value or the type of each field. This is expressed by supplying an actual parameter (a value) or a formal parameter (a variable) to an operation. If *age* is a variable of type integer and *married* is a variable of type boolean, then the tuple shown above can be read by

```
read("jones", ? age, ? married)
```

or read and removed by

```
in("jones", ? age, ? married)
```

where "?" denotes a formal parameter to be filled in when a tuple is matched. The variable *age* is assigned the value of the second field (31) and the variable *married* gets the value of the last field (true). Both operations try to find a matching tuple in TS. A tuple matches if each field has the value or the type passed as parameter to the operation. If several matching tuples exist, it is undefined which one is chosen. If there are no matching tuples, the operations **in** and **read** (and the invoking process) block until another process adds a tuple that does match (using **out**).

Linda has no operations that modify a tuple in place in TS. To change a tuple, one must first remove it from TS, then change it, and then put it back. Each **read**, **in**, and **out** operation is atomic: the effect of several simultaneous operations on the same tuple is the same as that of executing them in some (undefined) sequential order. In particular, if two processes want to remove the same tuple, only one of them will succeed and the other one will block.

The described properties of TS make it possible to support different programming paradigms. In addition to more traditional programming paradigms like the message passing paradigm [Carriero and Gelernter 1988], Linda's TS also supports the distributed data

structure paradigm. A distributed array, for example, can be built out of tuples of the form "[name, index, value]." The value of the element *i* of array *A* can be read into a local integer variable *X* with a simple read operation:

```
read("A", i, ? X)
```

To assign a new value *Y* to element *i*, the current tuple representing A[*i*] is removed first; then a tuple with the new value is generated:

```
in("A", i, ? tmp)
out("A", i, Y)
```

To (indivisibly) increment a global counter, the current tuple is removed from TS, its value is stored in a temporary variable and the new value is computed and stored in a new tuple:

```
increment(NameGlobal)
  char *NameGlobal;
{
  int tmp;

  in(NameGlobal, ? tmp);          /* "tmp" is a formal parameter */
  out(NameGlobal, tmp+1);         /* "tmp+1" is an actual parameter */
}
```

If two processes simultaneously want to increment the same global counter, the element will indeed be incremented twice. Only one process will succeed in doing the **in** and the other process will be blocked until the first one puts the new value of the global counter back into TS.

Linda provides a simple primitive (called **eval**) to create a sequential process. An earlier version of Linda provided constructs for parallel execution of a group of statements [Gelernter 1985]. We describe the current version here, which is based on C [Kernighan and Ritchie 1978]. Besides **in**, **out**, and **read**, it provides two other operations on TS: **inp** and **rdp**. **Inp** and **rdp** are similar to **in** and **read**, respectively. However, if there is no matching tuple in TS then they do not block, but immediately return false.

Linda has been implemented on the Bell Labs' S/Net [Carriero and Gelernter 1986], an Ethernet based VAX network, the iPSC hypercube [Bjornson et al. 1989; Gelernter and Carriero 1986], the Encore Multimax, the Sequent Balance, the Vrije Universiteit MC68020 based VME multiprocessor, and other machines. Besides these software implementations, there is also a hardware implementation [Ahuja et al. 1988; Krishnaswany 1988]. Implementation strategies are discussed in [Carriero 1987].

Linda has been criticized by many people in the computer science community. Some of the most persistent criticisms are: tuple space does not scale, it is not possible to implement Linda efficiently, and its address space is not structured. We consider most of these points of criticism unfair or not true. Linda has been implemented on many architectures and seems to scale until at least 64 nodes. A prototype Linda machine containing about 80 nodes is under construction. For almost all our Linda programs we get good performance on our shared memory machine, so, we think that Linda can be implemented efficiently. Although Linda contains no concepts for modularization yet, we see no fundamental reason why they could

not be added [Gelernter 1989]. In this paper, we will argue that Linda is an improvement over more traditional parallel languages, but that it could be improved on some major points.

4. EXAMPLE ALGORITHMS AND THEIR IMPLEMENTATION IN LINDA

Having described distributed data structures and Linda, we will now study two distributed data structures solutions and their implementation in Linda: a package (DIB) that supports a wide range of search strategies and the all-pairs shortest paths problem. The package illustrates how a reasonably complex problem can easily be solved with distributed data structures. The all-pairs shortest paths problem shows that the replicated worker style is not always applicable. Using a different programming style, however, one can still obtain almost linear speedup.

4.1. Distributed Implementation of Backtracking

Distributed Implementation of Backtracking (DIB), developed by Finkel and Manber [Finkel and Manber 1985; Finkel and Manber 1987], is a general-purpose package that supports different kinds of search algorithms, such as recursive backtracking, branch-and-bound, and alpha-beta. It runs on a network of computers, each with its own local memory, that communicate by exchanging messages. The application program needs only to specify the root of the search tree, the computation to be performed at each node, and how to generate children at each node. In addition, the application program may optionally specify how to synthesize values of tree nodes from their children's values and how to disseminate information, such as bounds, either globally or locally in the tree. Finkel and Manber have implemented DIB in Modula [Wirth 1977] directly on top of the Crystal multicomputer [DeWitt et al. 1984] and have reported results for the N-queens, traveling salesman, and alpha-beta problems.

We have redesigned and reimplemented a DIB-like package in Linda, called DIBL. With DIBL the same classes of problems can be solved. The implementation of DIBL, however, differs radically from DIB's. Instead of using operating system calls for message-passing and the concept of Concurrent Pools [Manber 1986], it uses the distributed data structures paradigm and replicated workers. DIBL also is functionally different from DIB. It does not provide fault-tolerance, since Linda does not provide it. However, DIBL provides several search strategies such as breadth-first search and depth-first search, which are not present in DIB.

We will first discuss DIBL in terms of distributed data structures. We will discuss our Linda implementation afterwards. DIBL uses the following distributed data structures:

- A set of nodes that form a tree. Each node contains a pointer to its parent, a counter for its generated children, a counter for its processed children, and a data structure containing application-dependent information. This set is used to build a search tree.
- A set of tasks (records). Each task contains a pointer to a node. This set is used for scheduling the tasks.
- A shared data structure that contains global application-dependent information. This can be used, for example, to store a bound for branch-and-bound applications.

When an application is started, a user-supplied application-dependent routine generates the root of the search tree. The root is put into the set of nodes and a task for the root is put into the set of tasks. A worker removes a task from the set of tasks and gets a copy of the corresponding node from the set of nodes. To expand the search tree, the worker repeatedly

calls an application-dependent routine that generates a child. For each child it adds a task to the set of tasks and a node to the set of nodes. In this way, the entire tree is generated and processed.

To propagate results to the parent after the child and its descendants have been processed, a node contains a pointer to the parent and a counter telling how many children have been processed. If all the children have been processed, a new task for the node is added to the set of tasks. Eventually this task is removed and an application-dependent routine can propagate the results from this node to its parent. Optionally, the application-dependent routine can specify that results are also propagated to siblings and their descendants (e.g., to distribute the search window in alpha-beta) or an update of global information (e.g., to improve the length of the best route found so far in the traveling salesman problem). In this way, information can be distributed in the search tree.

It is easy to extend DIBL to incorporate different search strategies. For example, we can easily implement a breadth-first strategy by using the following distributed data structures:

- A FIFO queue of tasks. Each task contains pointers to a node and to the next task.
- Two shared counters that identify the first and last task in the queue.

Each worker removes a task from the front of the task queue. If it generates new tasks, it puts them at the end of the task queue. Other strategies can be implemented using a different distributed data structure. Depth-first search, for example, can be implemented using a task stack.

DIBL is implemented in only 400 lines of C-Linda. The only significant problem we encountered using Linda to implement DIBL was the lack of a statement for expressing nondeterminism. To determine if a distributed application has finished, for example, one has to wait until one of two conditions becomes true: a worker has to wait until one of the other workers generates new work *or* until all other workers also are idle. Although in Linda, one can do an **in** on one pattern, this is not sufficient to express the stated condition; in this case the programmer needs an **in** on several patterns. A programmer could easily express this, however, with a statement such as a guarded command [Dijkstra 1975]. In Linda the obvious way to solve this problem is by polling using **inp** (see Fig. 1).

Initially, one of the workers sets the counter *ActiveWorker* to 1, generates the first task, and executes the forever-loop. After *ActiveWorker* is set to 1 all other workers increment *ActiveWorker* and also execute the forever-loop. Each worker removes a task from the *TaskBag*, performs the task and tries to remove a new task (using **inp**). When a worker cannot find work (**inp** fails), it decreases the global counter *ActiveWorker*. If the worker is the last active one, the program terminates. Otherwise, the worker enters a busy-wait loop and waits either for new work or for termination. In this way, a non-deterministic statement can be simulated. The disadvantages are that one has to use polling and non-standard Linda primitives (**inp** and **rdp**).

4.2. The All-Pairs Shortest Paths problem

In the All-Pairs Shortest Paths (ASP) problem, the shortest path from any node i to any other node j in a given graph must be found. A sequential, iterative, algorithm for ASP is given in Section 5.8 of [Aho et al. 1974]. The algorithm assumes that the nodes are numbered

```
BusyWaiting = FALSE;
in("ActiveWorker", ? count);                    /* increment ActiveWorker */
out("ActiveWorker", count+1);

while( TRUE ) {
  while( !inp("TaskBag", ? task) ) {            /* try to remove a task */
    if( BusyWaiting == FALSE ) {                /* get into busy-wait loop */
      in("ActiveWorker", ? count);              /* decrement ActiveWorker */
      out("ActiveWorker", count-1);
      if( count-1 <= 0 ) return();              /* everybody is busy waiting */
      BusyWaiting = TRUE;                       /* start busy-waiting */
    }
    else {                                      /* check if ActiveWorker is 0 */
      rd("ActiveWorker", ? count);
      if(count == 0) return();                  /* the work is done */
    }
  }
  if( BusyWaiting == TRUE ) {                   /* we have found a task */
    in("ActiveWorker", ? count);                /* we are going to work again */
    out("ActiveWorker", count+1);               /* increment ActiveWorker */
    BusyWaiting = FALSE;
  }
  Execute(task);                                /* start working */
}
```

Fig. 1. Simulating a non-deterministic statement.

sequentially (from 1 to n). During iteration k the algorithm finds the shortest path from every node i to every node j that only visits nodes in the set $\{1..k\}$. It does this by computing the matrix:

$$C'[i,j] = MIN(C[i,j], C[i,k] + C[k,j]) \quad (1 \leq i,j \leq n)$$

Thus, during iteration k, the algorithm simply checks if the (best) path from i to k concatenated with the best path from k to j is shorter than the best path from i to j found so far (i.e., during the first k-1 iterations).

Before the first iteration, such a path only exists if there is an edge in the graph from node i to node j. After the last iteration, the resulting path is the shortest path from node i to node j.

In the sequential algorithm, for each value of k a complete new matrix is computed (row k remains unchanged). Thus, an easy way to distribute the algorithm is to compute the values of all the rows of C' in parallel.

The parallel version using the replicated worker style consists of a set of workers that repeatedly perform the same task: updating all paths from one specific node i to all other nodes j (this corresponds to computing row i of C'). To update the paths for node i during iteration k, a worker needs to know the lengths of the current best paths from node i and node k to any other node. These $2 \times n$ values are stored in the work item.

It is straightforward to implement the algorithm in C-Linda. (The whole program takes approximately 80 lines of code.) Unfortunately, even with a large number of workers, the program is hardly faster than with one worker (maximum speedup is 1.5 by 4 processors). Analyzing the complexity of the algorithm shows that the actual work, updating the path for a specific node, takes order n operations and that managing the work also takes order n operations. Moreover, if taking work out of the *taskbag* is implemented using mutual exclusion (as in our implementation), the taskbag is a severe bottleneck and prevents the algorithm from *scaling* to many worker processes.

This communication overhead is not inherent to the ASP problem, but is due to the replicated worker style. For comparison, we also implemented another ASP algorithm that lets each processor manage a fixed part of the nodes; with p processors, each processor maintains n/p rows. This Linda program is more complex, but it is far more efficient and achieves an almost linear speedup if n is large. (With 8 processors we have measured a speedup of 7.4.) The development of the ASP program follows the design cycle for parallel programs proposed by the designers of Linda [Carriero and Gelernter 1988]. They identify three programming methods for parallel programming and describe how to transfer a parallel program from one method to another method. In the ASP example we started with a program using the replicated worker style (activity parallelism) and ended with a program using structure parallelism.

5. EVALUATION

Having illustrated how distributed data structures are used for writing distributed programs, we can evaluate the paradigm and its implementation in Linda.

5.1. The distributed data structure paradigm

There seems to be no consensus in the literature as to a standard for evaluating a parallel programming paradigm. Although a number of criteria have been proposed, no particular set has achieved the status of an undisputed yardstick [Burns et al. 1987]. We think, however, that the following list covers the most important aspects of a parallel programming paradigm: communication, mutual exclusion, synchronization, parallelism, fault-tolerance, and implementation issues. When compared with the better known message passing and shared variables paradigms, we will argue that the distributed data structure paradigm is easier to use and nonetheless has more expressive power.

Communication

As with shared variables, communication with the distributed data structure paradigm is very easy. In the message passing paradigm, however, communication is an important part of a parallel program. A programmer must address the questions "to whom do I send my message?" and "when do I send my message?". In contrast, processes interacting through a distributed data structure do not need to know each other's existence, and do not need to have overlapping lifetimes. The physical communication is completely hidden from the programmer.

Mutual exclusion and synchronization

As with message passing, but unlike shared variables, mutual exclusion is hidden from the programmer, because operations are conceptually serialized. Unlike message passing, however, all processes observe all events in the same order, because operations have immediate effect. Thus, the distributed data structure paradigm makes it possible to reason about the global state of the shared data, because at any instant the global state is unique, unambiguous, and identical for all processes.

Parallelism

If the distributed data structure paradigm is combined with the replicated worker style, writing distributed programs becomes very similar to writing sequential programs [Ahuja et al. 1986]. Once the data structure is chosen, and the algorithm for one worker is written, the complete parallel program is ready. Writing the worker program is similar to writing a sequential program, since the programmer does not have to deal with explicit parallelism. The examples in the previous section illustrate this; in each case the data structure and the worker algorithm define the complete distributed program.

The combination of the two, however, is not always applicable. The replicated worker style is unsuitable for ASP, for example. Our ASP example is structurally similar to the LU program described in [Carriero 1987], for which the replicated worker style works well. In the LU program, however, a single task performs n double-precision floating point multiplications and subtractions, instead of the n integer additions and comparisons required by an ASP task. For both programs, a task has $O(n)$ processing time and $O(n)$ communication overhead. For LU decomposition the constant factor of the processing time is relatively high, which explains why LU achieves a good speedup (with a small number of processors) and ASP does not.

Fault-tolerance

Although a fault-tolerant implementation of TS exists [Xu 1988], the distributed data structure paradigm provides no mechanisms to cope with hardware failures. For the message passing paradigm several fault-tolerant extensions exist, for example transactions [Lampson et al. 1981].

Implementation

The distributed data structure paradigm is hard to implement on an architecture without shared memory, because it does not reflect the underlying architecture. However, a clever implementation using an advanced compiler can execute programs efficiently [Carriero 1987; Bjornson et al. 1989].

Discussion

A final argument for the claim that the distributed data structure paradigm is a higher-level paradigm than message passing and shared variables is that it is easy to simulate message passing and shared variables with a distributed data structure. Implementing distributed data structures efficiently with message passing is very hard, however. In the message passing paradigm, a shared data structure must be encapsulated by a specific manager process that serializes all access to the data. In a distributed system the manager process can be a serious

bottleneck. In addition, a lot more process switches are needed. The shared variable paradigm is hard to implement efficiently on machines without shared memory, while the distributed data structure paradigm is less hard to implement, because of its more restricted semantics. In addition, the distributed data structure paradigm automatically performs mutual exclusion. Therefore, we think that the distributed data structure paradigm provides a higher-level of abstraction than the message passing or shared variables paradigms, yet admits an efficient implementation on both the distributed and shared-memory machines.

5.2. Support of Linda concepts for the distributed data structure paradigm

Above we evaluated the distributed data structure paradigm. Here, we will discuss the Linda concepts that support the distributed data structure paradigm. Again, there is no agreed upon yardstick for deciding how good these concepts are. But, we think that the following criteria are important.

1. The parallel and sequential concepts should be well integrated.
2. The language must have simple, unambiguous semantics.
3. Sufficient expressive power to write most programs should be available.
4. Efficient implementation should be possible.

Integration

The Linda concepts can be added to an arbitrary sequential language to get a parallel language. This has the advantage that users only have to learn about TS and its operations, which are easy to grasp. The disadvantage is that there are two ways to build data structures. Data structures can be built using tuples or using the structuring primitives of the base language. Consider, for example, the implementation of a 1K bit-vector. One has at least two choices: putting the complete bit-vector in one tuple or putting every bit in a separate tuple. The first choice limits concurrency. The second choice wastes memory. Thus the programmer is confronted with a dilemma.

A related problem is the choice between keeping a copy of the data structure local and storing indices to this data structure in TS, or keeping the complete data structure in TS. In the first case, **in**, **read**, and **out** are cheap, because only indices have to be moved from TS to the process executing the operations. But, an extra level of indirection is introduced, which makes programming harder. A matrix multiplication program, for example, can store copies of the two matrices on each processor and use the column and row numbers to identify tasks. Alternatively, it can store the rows and columns themselves in a task description. The only way to determine the best choice is to write programs for both cases.

Complexity and semantics

One of Linda's distinguishing properties are the simple semantics and the low complexity of its primitives. The Linda primitives and the working of TS can be fully described in a single page and are easy to master. Comparing the Linda concepts with, for example, the Ada concepts for parallel programming [U.S. Department of Defence 1983], demonstrates the clean semantics and the low complexity of the Linda operations and TS. (Although Ada intents to support a much wider range of applications than Linda.) Its support for parallel programming is more complex than Linda's. In addition to the rendez-vous mechanism, it supports

different classes of shared variables, all of them with a slightly different semantics. Complete books exist explaining the Ada primitives for parallel programming [Burns et al. 1987].

Expressive power

Linda's unique addressing method makes it easy to build data structures like arrays, bags, and sets. Lists and graphs are harder to build, however. For each list or graph a separate counter must be kept in TS to generate unique indices (e.g., consider the way queues are built in DIBL).

In contrast with the addressing method, the operations on TS are low-level. The **in** and **out** primitives provide concurrency control similar to the P and V semaphore primitives. As with P and V, the programmer has to be careful with **in** and **out**. Consider, for example, the start up phase of a typical Linda program. When an application is started (or finished), there is usually one worker process that behaves as a master (see Fig. 2). The master puts the shared variables in TS and becomes a normal worker. After the work is done, the master prints the results and cleans up TS. To decide if a worker is done, a global counter *ActiveWorker* is used (see Fig. 1). When a worker starts, it increments *ActiveWorker*. When there is no more work, it decreases *ActiveWorker* and waits until another worker generates work or until all workers have finished (*ActiveWorker* becomes 0).

```
main()
{
    if(master) {                          /* master is a parameter of the program */
        out("ActiveWorker", 0);           /* initialize global variables */
        out("job", 0);
        worker();                         /* master also becomes a worker */
        printresult();                    /* workers are finished; print results */
    }
    else
        worker();                         /* slave starts working */
}
```

Fig. 2. Structure of a typical Linda program.

This seems plausible, but it contains a serious error. Consider the following scenario. After the master has put *ActiveWorker* into TS, it is stopped, for example, because another process on the same processor is scheduled to run. A worker running on another processor increments *ActiveWorker* and starts looking for a job. Because there is no work (yet), it decreases *ActiveWorker*, finds out that the counter is zero, and thus terminates. Now the master resumes its work, dumps the first job in TS, and becomes an ordinary worker. After the work is done, it prints the results and then terminates. Seeing that the problem is correctly solved, the programmer is happy. When he measures the execution time however, he is disappointed; instead of linear speedup there is no speedup.

One way to solve this problem is to put the initial work in TS first and then the counter *ActiveWorker*. Thus, as is typical with low-level constructs, one subtle error and things go wrong. It is like programming in assembly language, only worse, because errors are irreproducible.

As illustrated above, not only must a programmer be careful with **in** and **out** to avoid race conditions, but the programmer can easily lose concurrency and thus increase the execution time of the program as well. When a tuple is removed from TS by an **in**, the tuple must be put back as soon as possible in order to make it available to other processes. Consider the usage of global information in DIBL. After a worker has removed a task from the set of tasks, it repeatedly calls an application-dependent routine that generates a child. One of the arguments for this routine is the global information, because the application may read or write the global information. An unsuspecting programmer would write the following Linda code for expanding the search tree (see Fig. 3).

```
in("GlobalInfo", GlobalInfo);                    /* take GlobalInfo from TS */
do {
    GenerateChild(&done, &child , &GlobalInfo);  /* generate child */

    /* code for the generated child */
} while(done == FALSE);                          /* are all children generated? */
out("GlobalInfo", GlobalInfo);                   /* put GlobalInfo back */
```

Fig. 3. Processing of *GlobalInfo*.

Another worker that also wants to read the global information is blocked by the **in**. As *GenerateChild()* is likely to be a time-consuming call, this would lead to a severe reduction in performance. A better implementation would do the management of the global information in a cumbersome manner (see Fig. 4). As can be seen, this implementation is rather clumsy, but gives a better performance.

```
read( "GlobalInfo", GlobalInfo);                 /* make copy of GlobalInfo */
TmpGlobalInfo = GlobalInfo;
do {
    GenerateChild(&done, &child , &GlobalInfo, &globalupdate); /* generate child */

   /* code for the generated child */
} while(done == FALSE);                          /* all children generated */

if(globalupdate == TRUE) {                       /* is copy of GlobalInfo changed? */
    /* Yes, change GlobalInfo. */
    in("GlobalInfo", &GlobalInfo);
    GlobalUpdate(GlobalInfo, TmpGlobalInfo);     /* check and update */
    out("GlobalInfo", GlobalInfo);
}
```

Fig. 4. Processing of *GlobalInfo* in a clumsy, but efficient way.

In addition to being low-level, the operations are inflexible; for a single tuple the operations provide mutual exclusion automatically, but for a set of tuples the programmer must do the mutual exclusion. For example, comparing two distributed sets is difficult (see Fig. 5). One needs tuples that provide mutual exclusion for these sets. When the critical region is

entered, all the elements of set1 need to be removed by an **in**, before they can be used to check if they also are a member of set2. Then, the elements can be stored back into TS and finally the mutex must be made available again.

Another problem with flexibility in Linda is the absence of a statement to express non-determinism. That is, a programmer is able to do an **in** on a single pattern, but not on multiple patterns. Some implementations provide the non-blocking statements **inp** and **rdp**, but there is no implementation that provides a general nondeterministic statement. Such a statement would be useful, as argued in the DIBL example (see Fig. 1). A statement similar to a guarded command [Dijkstra 1975] would make Linda more powerful, although it would make the implementation more complex.

```
Compare( set1, set2)
   char *set1, *set2;
{
   SetType S1;

   result = TRUE;
   in("mutex", set1, size1);                    /* enter critical region */
   in("mutex", set2, size2);
   if( size1 == size2) {                        /* Are number of elements equal? */
      /* Yes, check if the sets are really equal */
      for( i=0; i < size1; i++) {               /* collect all the set elements */
         in(set1, ? e);                         /* take element of set1 from TS */
         S1 = S1 + { e };                       /* add it */
         if( !rdp(set2, e) ) {                  /* compare */
            result = FALSE;
            break;
         }
      }
   }
   else result = FALSE;
   ...  /* Code to put set1 back into TS */
   out("mutex", set1, size1);                   /* make the mutexes available */
   out("mutex", set2, size2);
   return( result );
}
```

Fig. 5. Comparing two distributed sets.

Implementation

TS has been implemented successfully on many different architectures. The programs written in Linda have shown good speedups on these architectures. Making a Linda implementation efficient, however, requires advanced compiler techniques [Carriero 1987].

It is impossible to implement garbage collection on TS, because tuples are addressed by contents. A garbage collector can never decide if a tuple is going to be referenced in the future. So the programmer is responsible for throwing away tuples that will not be used any more.

5.3. C-Linda

The Linda primitives have been embedded in several sequential languages (e.g., C, FORTRAN, Modula-2). In general, there can be conflicts between the host language and Linda. In C-Linda, for example, passing structured arguments to the Linda primitives is awkward. To pass an array as an argument to **out** the following C-Linda code is needed:

```
long X[10];
LINDA_BLOCK ARGUMENT;     /* LINDA_BLOCK is a structure defined
                             by the Linda compiler. */

ARGUMENT.data = X;
ARGUMENT.size = 10 * sizeof(long);

out( ARGUMENT );
```

To pass the array *X* to an **out**, an extra data structure has to be declared: a "LINDA_BLOCK." After initializing this structure, the pointer to the "LINDA_BLOCK" is given to **out** to get *X* in TS. Clearly, C-Linda could be more programmer friendly†.

6. CONCLUSIONS

We have studied a paradigm, distributed data structures, that achieves a higher level of abstraction than existing paradigms for parallel programming. In addition to this paradigm, we studied a programming style: the replicated worker style. We presented two examples illustrating the distributed data structure paradigm and the replicated worker programming style. The distributed backtracking package shows how a complex problem can easily be solved with distributed data structures. The all-pairs shortest paths problem shows that the replicated worker style of programming is flexible, but may cause efficiency problems. These problems can be solved by transforming the replicated worker program into a program using structure parallelism.

Although Linda has many unique and clear concepts for parallel programming, it also has some significant flaws: it is unclear how distributed data structures should be built; the operations on TS are too primitive and low-level: concurrency control is inadequate (Linda's operations are low-level and do not support a general statement to control nondeterminism), and automatic mutual exclusion is only provided on single tuples. Therefore, we think that Linda provides a too-low level implementation of the distributed data structure paradigm, but that it still gives more support for parallel programming than languages based on the message passing and shared variables paradigms.

Looking at the development of programming languages for sequential programming, we see that the earlier languages like assembly languages do not put any constraints on the programming style, resulting in programs that are difficult to understand and error prone. Modern languages like Modula-2 [Wirth 1985] and Smalltalk [Goldberg and Robson 1983] force a programming style upon the programmer, leading to clear programs. We think that

† Modula-2/Linda does not have this problem. Passing general data structures like graphs, however, is still a problem [Borrman et al. 1988].

the development of languages for parallel programs will go through a similar development. In our view, a step in this direction can be made with Linda and the distributed data structure paradigm.

We are working on a different approach to the distributed data structure paradigm, Orca [Bal and Tanenbaum 1988]. In Orca, programmers can define abstract distributed objects, along with high-level operations on these objects. This method eliminates the problems that we have pointed out in Linda. At the moment we have prototypes running on a multi-processor with shared memory, on a number of processors connected by Ethernet [Bal et al. 1989a], and an implementation on top of Amoeba [Mullender and Tanenbaum 1986] on machines without shared primary memory.

ACKNOWLEDGEMENTS

We want to thank Erik Baalbergen, Dick Grune, Susan Flynn Hummel, Robbert van Renesse, and the referees for their valuable suggestions and the critical reading of this paper. Also, we would like to thank the Linda group at Yale, Rob Bjornson, Nick Carriero, David Gelernter, and Jerry Leichter for their comments.

REFERENCES

Aho, A.V., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA. (1974).

Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *IEEE Computer*, Vol. 19, No. 8, pp. 26-34 (Aug. 1986).

Ahuja, S., Carriero, N.J., Gelernter, D.H., and Krishnaswamy, V., "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Trans. Computers*, Vol. C-37, No. 8, pp. 921-929 (1988).

Andrews, G.R., Olsson, R.A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G., "An Overview of the SR Language and Implementation," *ACM Trans. Progr. Lang. Syst.*, Vol. 10, No. 1, pp. 51-86 (Jan. 1988).

Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., "A Distributed Implementation of the Shared Data-object Model," *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (Oct. 1989a).

Bal, H.E., Steiner, J.G., and Tanenbaum, A.S., "Programming Languages for Distributed Computing Systems," *Comp. Surv.* (Sept. 1989b).

Bal, H.E. and Tanenbaum, A.S., "Distributed Programming with Shared Data," pp. 82-91, Miami, FL. (Oct. 1988).

Bjornson, R., Carriero, N., and Gelernter, D., "The Implementation and Performance of Hypercube Linda," Report RR-690, Yale University, New Haven, CT (March 1989).

Borrmann, L., Herdieckerhoff, M., and Klein, A., "Tuple Space Integrated into Modula-2, Implementation of the Linda Concept on a Hierarchical Multiprocessor," Cambridge University Press.

Burns, A., Lister, A.M., and Welling, A.J., "A Review of Ada Tasking" in *Lecture Notes in Computer Science 262*, Springer-Verlag, Berlin (1987).

Carriero, N., "The Implementation of Tuple Space Machines," RR-567 (Ph.D. dissertation), Yale University, New Haven, CT (Dec. 1987).

Carriero, N. and Gelernter, D., "The S/Net's Linda Kernel," *ACM Trans. on Comp. Syst.*, Vol. 4, No. 2, pp. 110-129 (May 1986).

Carriero, N. and Gelernter, D., "How to Write Parallel Programs - A Guide to the Perplexed," Report RR-628, Yale University, New Haven, CT (May 1988).

Carriero, N. and Gelernter, D., "Linda in Context," *Commun. ACM*, Vol. 32, No. 4, pp. 444-458 (April 1989).

Carriero, N., Gelernter, D., and Leichter, J., "Distributed Data Structures in Linda," *Proc. 13th ACM Symp. Princ. Progr. Lang.*, pp. 236-242, St. Petersburg, Fl. (Jan. 1986).

Dijkstra, E.W., "Guarded Commands, Nondeterminacy and Formal derivation of Programs," *Comm. ACM*, Vol. 18, No. 8 (Aug. 1975).

D.J., DeWitt, R., Finkel, and M., Solomon, "The Crystal multicomputer: Design and implementation experience.," Report #553, Computer Sciences Department, University of Wisconsin., Madison, Wisc (Sept. 1984).

Finkel, R. and Manber, U., "DIB - A Distributed Implementation of Backtracking," *Proc. 5th Int. Conf. on Distr. Comp. Syst.*, pp. 446-458 (May 1985).

Finkel, R. and , U. Manber, "DIB - A Distributed Implementation of Backtracking," *ACM Trans. Progr. Lang. Syst.*, Vol. 9, No. 2, pp. 235-256 (April 1987).

Gehani, N.H. and Roome, W.D., "Concurrent C," *Softw. Prac. Exper.*, Vol. 16, No. 9, pp. 821-844 (Sept. 1986).

Gelernter, D., "Generative Communication in Linda," *ACM Trans. Progr. Lang. Syst.*, Vol. 7, No. 1, pp. 80-112 (Jan. 1985).

Gelernter, D., "Multiple tuple spaces in Linda," *PARLE 89*, Eindhoven, The Netherlands (June 1989).

Gelernter, D. and Carriero, N., "Linda on Hypercube Multicomputers," *Proc. 1985 SIAM Conf.*, pp. 45-55, Knoxville, Tenn (1986).

Geschke, C.M., Jr., J. H. Morris, and Satterthwaite, E.H., "Early Experience with Mesa," *Commun. ACM*, Vol. 20, No. 8, pp. 540-553 (Aug. 1977).

Goldberg, A. and Robson, D., "Smalltalk-80: the Language and its Implementation," Addison-Wesley, Reading, Mass. (1983).

Hansen, P. Brinch, "The Programming Language Concurrent Pascal," *IEEE Trans. Softw. Eng.*, Vol. 1, No. 2, pp. 199-207 (June 1975).

Hansen, P. Brinch, "The Design of Edison," *Softw. Prac. Exper.*, Vol. 11, pp. 363-396 (1981).

Hoare, C.A.R., "Communicating Sequential Processes," *Commun. ACM*, Vol. 21, No. 8, pp. 666-677 (Aug. 1978).

Kernighan, B.W. and Ritchie, D.M., "The C Programming Language," Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978).

Krishnaswany, V., Ahuja, S., Carriero, N., and Gelernter, D., "The Architecture of a Linda Coprocessor," *15th Int. Conf. on Computer Architecture*, pp. 240-249, IEEE (June 1988).

Lampson, B.W., "Atomic Transactions," pp. 246-265 in *Distributed Systems - Architecture and Implementation*, ed. B.W. Lampson, Springer-Verlag, Berlin and New York (1981).

Manber, U., "On Maintaining Dynamic Information in a Concurrent Environment," *SIAM J. Comput.*, Vol. 15, No. 4, pp. 1130-1142 (Nov. 1986).

Mullender, S. J. and Tanenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer J.*, Vol. 29, No. 4, pp. 289-300 (March 1986).

Tanenbaum, A.S. and van Renesse, R., "Distributed Operating Systems," *Comp. Surv.*, Vol. 17, No. 4, pp. 419-470 (Dec. 1985).

U.S. Department of Defence, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A (Jan. 1983).

Wirth, N., "The Programming Language Pascal," *Acta Informatica*, Vol. 1, No. 1, pp. 35-63 (1971).

Wirth, N., "Modula: A Language for Modular Multiprogramming," *Softw. Prac. Exper.*, Vol. 7, No. 1, pp. 3-35 (1977).

Wirth, N., "Programming in Modula-2," Springer-Verlag, Berlin (1985).

Xu, A.S., "A Fault-Tolerant Network Kernel for Linda," Report MIT/LCS/TR-24, MIT, Cambridge, Mass. (Aug. 1988).

TUMULT-64: A REAL-TIME MULTI-PROCESSOR SYSTEM

Pierre G. Jansen,
Gerard J.M. Smit

Twente University
Department of Computer Science
P.O. Box 217
7500AE Enschede
The Netherlands

ABSTRACT

Tumult[1] is a modular extendible multi-processor system (MIMD) with distributed memory. The processors communicate via a high performance switching network. A distributed real-time operating system has been designed and implemented, offering high performance communication facilities at application level. Processes can connect dynamically to communication links after which they may pass messages or call remote procedures. This article gives a brief description of the system and emphasizes architectural - and communication aspects.

Key-words
MIMD-computer, store-forward switching network, distributed operating system, real-time, inter-process communication, communication ports, message passing, remote procedure.

1. Introduction

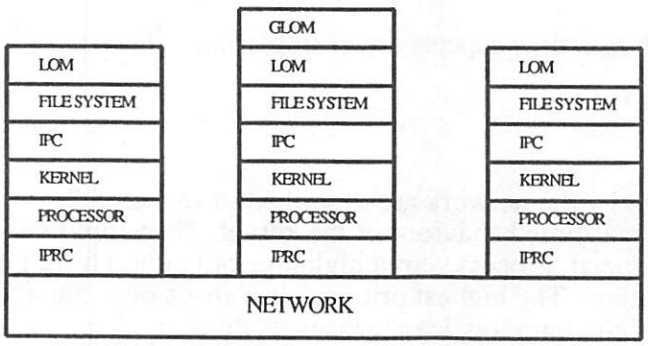
The aim of the Tumult project is to pursue research in the field of parallel architectures, in particular in the field of *multi-processor* systems. A small family of modular extendible multi-processors has been designed. Two systems are operational now, viz. Tumult-6 and Tumult-16. A third system, Tumult-64, is under development.
Tumult-6 merely served as a prototype system for Tumult-16. It has a maximum of six Motorola M680X0 processors, all inter-connected via a shared VME-bus [VME 85]. The operating system has been written in Modular Pascal [Bron 82].
Tumult-16 [Jansen 85] connects a maximum of 16 M680X0 processors to a store-forward communication network. This network has the following characteristics:

- It has a ring-structure which is modular extendible.
- The total capacity of the network is 20 Mbytes/Sec (worst case) (and 40 Mbytes/Sec typical).
- All nodes send or receive messages simultaneously, provided that the total sum of all messages offered does not exceed the total capacity of the network.

Tumult is used as a prototype for high performance applications.

[1] Tumult stands for Twente University MULTi processor. Tumult16 is developed in co-operation with the Dr. Neher Laboratories (Dutch PTT), Oce Nederland B.V. and the Twente University.

A major reason for the design of a multi-processor is to obtain a high performance. However the overhead caused by the communication control, strongly influences this performance. A recent study [Scott 87] showed that most multi-processors have a communication control overhead in the order of several milliseconds for simple message passing or a remote procedure call. The V kernel [Cheriton 85] for instance, which puts great emphasis on speed, requires 1.64 milliseconds for a request-reply within a machine, and 3.1 milliseconds between machines (SUN workstations and a 10 Mbits Ethernet). Due to the characteristics of Ethernet only one communication is active; a following communication request can only be honoured after the termination of the preceding request. In Tumult-16 **all** communications can be active simultaneously. For *each* communication it holds that the *total* control overhead, for the transfer of a message between processes on different nodes, is less than 1 millisecond (M68020 16 Mhz version). This overhead will be reduced considerably by implementing part of the communication control in hardware. For Tumult-64 our target is an overhead in the order of magnitude of 100 microseconds. Tumult-64 is an extension of Tumult-16 in which a maximum of 64 M680X0 processors are interconnected.
In the following sections we will describe Tumult-64 with its improved design decisions. First a general overview of the system is given, subsequently architectural and communication aspects are highlighted.
An extensive survey of multi-instruction stream computers (MIMD) is given in [Hockney 85]. In this taxonomy Tumult-64 could be characterized as an MIMD computer of which the processing elements (nodes) are interconnected via a bi-directional ring structure. (Ring structures have been used in several architectures in different variants, for instance ZMOB [Rieger 80] and Cyberplus of the CDC had a ring structure.)
The current system is not distributed geographically because of performance reasons. The network however, could be implemented as a serial (token passing) ring, which is appropriate for geographical distribution. In [Scholten 87] a proposal for a serial data link for the Tumult system is described.
The distributed real-time operating system is written in Modula-2 [Wirth 85]. Its characteristics are described in section 4. Some other interesting distributed operating systems are: Eden [Black 85], V-kernel [Cheriton 85], Charlotte [Artsy 87], DEMOS/MP [Miller 87], Accent [Rashid 81]. As far as we are informed they do not have real-time properties.



GLOM = GLOBAL MANAGER
LOM = LOCAL MANAGER
IPC = INTER PROCESS COMMUNICATION
IPRC = INTER PROCESSOR COMMUNICATION

Fig. 1. The layers of the tumult system.

In the systems mentioned, some form of message passing or remote procedure call [Birrel 84] or both is used. For a general review of communication primitives we refer to [Tanenbaum 85]. The communication primitives of Tumult are described in the sections 3 and 4.

2. Overview of Tumult

The purpose of this section is to give a quick overview of Tumult-64. Fig. 1 shows the architecture.

The hardware includes a modular extendible *interconnection network* which interconnects up to 64 processor nodes. This network transfers single data- or control words.
The *inter-processor communication layer* transfers records of arbitrary length from any to any node processor.
Section 3 describes the hardware in more detail.

The distributed operating system is written in Modula-2 [Wirth 85]. It is structured according to the layers shown in fig.1.

An efficient *real-time multi-tasking kernel* [Luttmer 87] runs at each processor node. It offers primitives such as memory allocation, process creation and -termination, deadline scheduling, interrupt handling, and (distributed) exception handling [Bron 84].
The *interprocess-communication* (IPC) layer allows for dynamic creation and deletion of logical communication links [Ribbers 87], shortly referred to as links. A link is a flexible communication structure, which can be adapted to the current communication needs of the system. It is believed to contain new elements for IPC and it was introduced for the first time in Tumult-16 (slightly different from the description in this paper). Links support Message Passing (MP) and Remote Procedures (RP) [Birrel 84]. Section 4.1 describes the logical communication link in more detail.
A *distributed file system* allows files and devices to be distributed over the nodes transparently [Langen 87]. The file system inherits its dynamic behaviour from the IPC-primitives and allows for the dynamic installation of devices and files.
The *Local Manager* (LM) receives, interprets and executes commands from the Global Manager (GM), such as load - , start - , or terminate a (sub) tasks, and it collects local status information.
The *Global Manager* interprets commands from console for the execution of (parallel) tasks. A task includes one or more different sub-tasks. The GM distributes the sub-tasks over the nodes and orders the LMs to execute them. The GM also collects global status information.

2.1. Real time aspects

There are four levels at which real-time aspects are of importance. These levels are:
- the network
- the kernel
- the IPRC
- the IPC

The real-time facilities offered by the network are described in section 3.3.
Much attention is paid to the real-time behaviour of the kernel. Therefore three different priority classes of processes are introduced. Processes of a higher priority class have precedence over processes of a lower priority class. The highest priority class maps onto the traditional interrupt-handlers, the second priority class onto low level processes that can never be stopped but always run to completion. These processes are called impulses. As interrupts, impulses have a static priority and are scheduled according to them.
The lowest priority class maps onto threads which are scheduled according to their deadlines. The priority of a thread is determined by the shortest deadline of all threads that are waiting for resources which are owned by this thread. (If no threads are waiting, the own deadline determines the priority.) In this way threads can be hurried in order to release there resources in time.
In particular by the use of impulses, we have been able to reduce the maximum interrupt disable time to 6 microseconds in a systematic way.
The real-time aspects of the IPRC and the IPC are still under investigation; the kernel is described in detail in [Luttmer 89].

3. The hardware

The hardware of Tumult-64 includes a maximum of 64 processor nodes which can communicate via a fast modular extendible switching network. Each processor-node consists of a network communication module and one or more processor modules. The network communication modules are connected in a ring topology, thus forming a communication network.

3.1. The network

The main task of the network is to *transfer single words* from any sender node to any receiver node. The network has a bi-directional ring topology.
According to [Feng 81] the network can be characterized as a synchronous store-forward network. It uses a demand assignment access mechanism. The network offers a high performance communication service with error detection, error correction, and

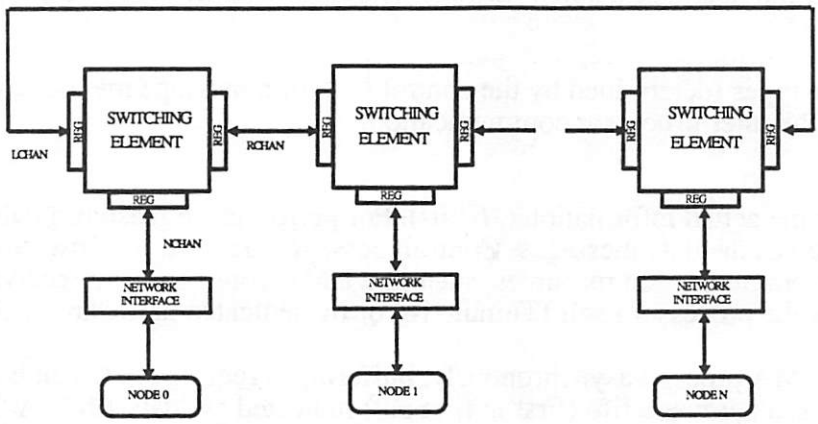


Fig. 2. A ring structured network of switching elements.

real-time access.
The network may optionally work in a real-time mode, in which a certain bandwidth can be claimed by a sender node (see section 3.3).
The basic building block of the network is a switching element (SE) 1[Jansen 80], arranged in a ring topology (fig. 2). They can switch messages in parallel (29 bits) via communication channels. Each SE has a full duplex communication channel to the node processor (*nchan*). Furthermore it has two half duplex communication channels *(rchan* and *lchan*) to the two adjacent switching elements.
A message has four fields:
- a destination address field (6 bits);
- a control field (5 bits);
- an information field (16 bits);
- a parity field (2 bits).

A SE has a unique address. If a message arrives from the *lchan* or *rchan* and the destination address field matches this unique address then it is transferred *to nchan*. Otherwise the message is transferred to *rchan* respectively *lchan*. When a processor is ready to send, a message is offered via *nchan* to a SE. The SE puts the message in the first arriving empty slot (demand assignment access mechanism). If two messages request the same output channel simultaneously, priority is given to the passing traffic.
All switching elements are synchronized with a global clock.
Each switching element has three output registers, one for left traffic, one for right traffic, and one to the local interface. At every odd tick of the global clock (running at 10 Mhz) all left traffic

is forwarded one node and at every even tick all right traffic is forwarded one node. Traffic to the local connection register is forwarded every tick.

A ring topology is not optimum with respect to fault tolerance and network diameter. The strength of this topology however is its simplicity, which allows a high bandwidth (20 Mbytes per second, worst case) and a small latency (100 nano seconds) between two neighbouring nodes. Because of this small latency, the total latency between two arbitrary (non neighbouring) nodes is relatively small. (The network latency has a modest contribution to the communication protocol overhead: it is less than 15 percent). This compensates sufficiently for the relatively large network diameter of the ring.
The fault tolerance aspect is considered in section 3.4.

3.2. Inter-Processor communication

The inter-processor communication transfers *variable length records* from any sending node to any receiving node.

Two low level message types (determined by the control field of a message mentioned earlier) are used to implement the inter-processor communication:
- data messages,
- control messages.

Data messages contain the actual information (16 bits). For performance reasons a DMA controller sends or receives the data messages. Control messages are used for flow-control as well as for requesting communication resources, such as a DMA controller or a receiver buffer. They are exchanged by the processor itself (Tumult-16) or by dedicated hardware (Tumult-64).

Because network and DMA proceed asynchronously, buffering is required between both. The buffer is implemented as a hardware fifo (first in first out) indicated as "data-fifo". A "sliding window" flow-control mechanism, implemented in hardware, is used to prevent overflow.

Hardware flow-control

A window counter at the sender side keeps track of a window indicating the available data-fifo space at the receiver side. It is initialized to the maximum buffer space. Each time the node (i.e. the DMA controller) transfers a message to the network the window is decremented. The transmissions stops when the window reaches zero. A counter at the receiver side records the number of messages received from the data-fifo. Each time when a fixed number of messages, say N, is received, a token message is returned to the sender. On reception of this token, the window counter is incremented by N.

A record is transferred to a buffer (process) *by a send-record* protocol and from a buffer by a *receive-record* protocol. Sender, buffer and receiver may reside on any node of the system.

The send-record protocol

If sender and buffer are residing at different nodes, the following protocol is used:
The sender transmits a "send-request" control message to a buffer where the needed resources (buffer space and a receiver DMA with data-fifo) are claimed, after which the control message "ready-to-receive" is returned. After the actual transfer of the record from sender to buffer, the control messages "send-ready" (from sender-side) and "receive-ready" (from buffer-side) are exchanged in order to acknowledge a transfer of the record. The claimed resources are released and the buffer administration is updated.

The receive-record protocol

The receive-record protocol proceeds similar to send-record protocol. After the exchanging of the control messages "receive-request" (from receiver-side) and "ready-to-send" (from buffer-

side) a record is transferred from buffer to receiver. The control messages "send-ready" (from buffer-side) and "receive-ready" (from sender side) conclude the record transfer.

Synchronous communication
If the size of the buffer is zero, the data exchange between sender and receiver is synchronous. The (empty) buffer (process) registers the reception of a matching "send-request" - "receive-request" pair. It notifies the receiver of such an event, which in turn notifies the sender after which the sender may send a record directly to the receiver.

For performance reasons **all** control messages are implemented as single words.
The record transfer protocol offers a virtual circuit service: it guarantees that a record, which has been sent, will always be received. (Since all the necessary resources have been claimed before.) This is in contrast with a datagram service, where a record is sent, regardless whether it can be received. If a record cannot be received, a re-send is requested. Under heavy load conditions a datagram service may cause trashing effects, caused by frequent re-sends to full buffers. This can not be tolerated in real-time systems.

The time needed to transfer the actual data depends on the DMA speed and is, in our case, in the order of magnitude of some bytes per microsecond. In the Tumult-16 the overhead of the inter-processor protocol (due to network latency, interrupt handling, process-switching, queueing, tests, etc) is in the order of 1 milliseconds (For a M68020 running on 16 MHz), a time in which also several Kbytes (depending on the DMA used) could have been sent. For small messages the protocol overhead overrules the transfer time.
To reduce this overhead, special hardware has been designed, which handles the most time critical part of the communication protocol. It provides facilities for sending and receiving of records concurrently to the node processor. The node processor may give a send- or receive order to its communication hardware, which will perform the appropriate actions and, when finished, signals the node processor by an interrupt.

3.3. Real-time aspects
The Tumult system has been designed with real-time applications in mind. The communication hardware and the operating system must be able to respond to a certain action before a specified deadline. This has its consequences for the kernel [Luttmer 87] and the communication.
Extra hardware has been added to each node to support real-time requirements.
A precise and consistent local knowledge of global time is useful for synchronization- and scheduling purposes. Therefore each network interface is equipped with a local clock that keeps the global time. All local clocks are reset at start up time and synchronously advanced by the global clock so that every clock shows the same time.

A second hardware feature is a simple mechanism to guarantee access to the network. With this mechanism bandwidth can be claimed in the following way:
The network uses a demand assignment access mechanism: a node has to wait until an empty message slot passes the switching element. In Tumult-64, slots can be claimed (and released; there is a claim-bit in the control field of a message) for exclusive use by a node. Empty slots, which are not claimed, may be used by any node. If N nodes are connected to the network, N slots are supported. Since the total capacity of the network is 20 Mbytes per second, a claim on a slot guarantees a bandwidth of 20/N Mbytes per second.

3.4. Fault tolerance
Reliability is an important aspect of multi-processor systems. Since our network has a ring topology, it would be too vulnerable without using fault tolerant techniques. Therefore, it supports bi-directional traffic in order to re-route messages in case of permanent stuck-at faults. Furthermore transient (e.g. due to noise) and intermittent (e.g. due to bad contacts) failures in a

message are detected by a parity checker at each switching element. Mutilated messages are immediately discarded by the switching element that detects the fault.
This introduces the disappearance of (mutilated) messages. However *received* messages are guaranteed to be correct with a high probability. How we cope with missing messages is described in the sequel.

Missing data messages
Missing data messages will cause an *underflow* situation in the send record protocol at the receiver side. *Underflow* is detected if:
- a "send ready" control message is received. (This guarantees that data messages cannot be under way any more.)
- the DMA is still waiting for more messages and
- the data fifo is empty.

If this happens a re-transmission of the record is requested. (Note that *overflow* is detected if a *send-ready* control message is received, the DMA is ready but the DMA fifo is not yet empty.)

Missing control messages
In order to protect against the loss of control messages an *alternating bit* protocol between nodes is used. Control messages are either request- or acknowledgement messages. They contain a message count (implemented as a mod 2 counter in a bit field) which serves as an identification for the message. A request may be sent to an arbitrary node, if the acknowledgement of the previous request to this node has been received. At the sending of a request, the count is increased by one and a time-out is set. The time-out is re-set at reception of the according acknowledgement. If the acknowledgement does not come in, the last message with its original count is re-sent.
On reception of a request, its count is compared to the precedingly received request. If they are equal, it is concluded that the message has been received before and only an acknowledgement with the same count is returned. If they are not equal, the requested action is taken and an acknowledgement with according count is returned.

(If the time-out expires too early, following requests may all cause acknowledgements. Only the first incoming acknowledgment has effect; all the others are neglected.
After registration of the first incoming acknowledgement of request N, the following request N+1 may be sent. Because the network is order preserving, possible sub-sequent acknowledgements of request N are coming in (and are neglected) before the acknowledgement(s) of request N+1.)

4. Inter-Process Communication
The Inter-Process Communication layer performs the communication between processes. For this purpose an administration of the current inter-process communication topologies is kept in a distributed data-base. These topologies are called *links* and they are system wide. Any process in the system with a valid access key may connect to such a link. Processes connected to the same link may communicate.

4.1. Links
Consider the set N of all processes of the system. A link L(S) is a communication topology on any subset S of N. A process p is said to be connected to a link L(S) if p is an element of S. Processes can be connected to a link for a particular link service (e.g. sending of a message or receiving of a message). Processes which are not connected to the same link can not communicate with each other.

Links can be created by processes *dynamically*. (After its creation the link contains the empty subset of processes.)
The subset S of processes connected to link L(S) can be changed dynamically, i.e. processes can be added to-, or removed from the original subset by *connect* or *disconnect* commands, which

extend implicitly the link topology. In this way links are adapted to the current communication needs of the system.
Links offer transparency of locality for connected processes: processes do not need to know each others physical location in order to communicate. (During the connection phase all relevant link information is exchanged between the connecting process and the distributed link administration.)
Links are identified by a system wide unique name. This link name serves as a key (or password) for the use of the link. A process is connected to a link via *a port*, which is essentially a local reference to a link. Ports may be extended with capability features for protection reasons [Mullender 85].
There are two types of links:
- Message Transfer (MT) links
- Remote Procedure (RP) links

4.2. Message Transfer links

A message transfer link L(S) is a uni-directional "many to any" communication topology, which supports the transfer of typed records between a subset S of processes. "Many to any" implies that a message, sent by a connected sender (many senders may be connected to the same link and may send simultaneously), is received by any (exactly one) of the connected receivers.
In order to improve the effective use of the offered parallelism a MT link may support buffered communication. The number of buffers can be defined at application level. This offers the possibility to avoid system deadlock due to a lack of buffer capacity in circular communication structures (see also [Kessels 80] and [Waumans 81]).
A MT link with some connected processes is shown in fig. 3.
It can be created by any (exactly one) process in the system in the following way:

```
CreateMTLink ('LinkKey', <record type>, NrOfBuffers);
```

in which "LinkKey" is the unique key to the link. <record type>

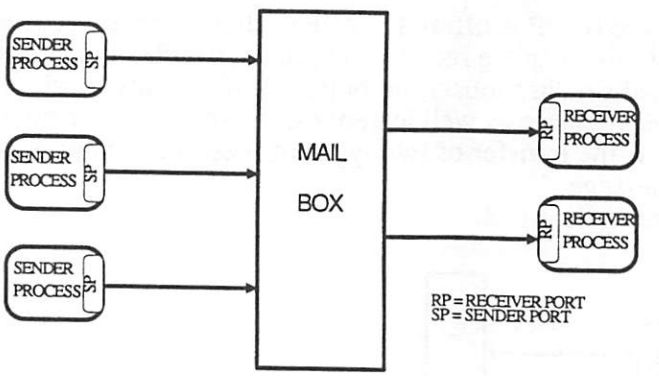


Fig. 3 A Message Transfer link with connected processes.

gives a straightforward type indication of the records to transfer. In the following this type is referred to as the "link component type". "NrOfBuffers" determines the number of buffers (of <record type>). These buffers are allocated at the residential node of the calling process. If "NrOfBuffers" is zero, all communication via this link will proceed synchronously as in CSP [Hoare 81]: it is guaranteed that after a successful send action the message(=record) is received.
If no processes are connected to the link (any more), it can be removed by its creator by:

```
RemoveMTLink ('LinkKey')
```

A process makes a connection to a link with identification "LinkKey" by the following procedure:

```
Connect (xp,'LinkKey');
```

A connect action blocks if a link is not (yet) created; when the action is finished a link has been created and the calling process is connected to it. "xp" is either a sender-port or a receiver-port. These ports have to be declared by sender- or receiver process in the following way respectively:

```
VAR xp: SenderPort;
VAR xp: ReceiverPort;
```

A process which is connected to a link via either a sender- or a receiver port "xp", can disconnect with:

```
Disconnect (xp);
```

If a process is connected to a link via sender port "sp" it may send a value "f" with the following procedure:

```
Send (sp, f);
```

The type of "f" must correspond to the link component type.
If a process is connected to a link via receiver port "rp" it may receive a message in a variable "a" with:

```
Receive (rp, a);
```

The type of "a" must correspond to the link component type.

A MT link guarantees the following properties:

1. Records are considered to be atomic in the sense that a record is received as it was sent and that no other records can intervene.
2. Records sent by different senders via the same link are received in any order by any receiver.
3. Records received by a receiver and originated from the same sender are received in the order in which they are sent.
4. If the number of link buffers is zero, the procedures "send" and "receive" are synchronized.

4.3. Remote Procedure links

Within Tumult a rudimentary type of RP is offered: a caller (client) sends a request to a callee (server) and then waits for the reply; a callee receives a request, handles it and sends the reply. The communication is performed synchronously; no buffering in the link is admitted.
The same link is used to transfer requests as well as replies. Therefore, at the client - as well as at the server side, a port must allow the transfer of two types of messages in opposite directions: a request-message and a reply-message.
An example of an RP-link is shown in fig. 4.

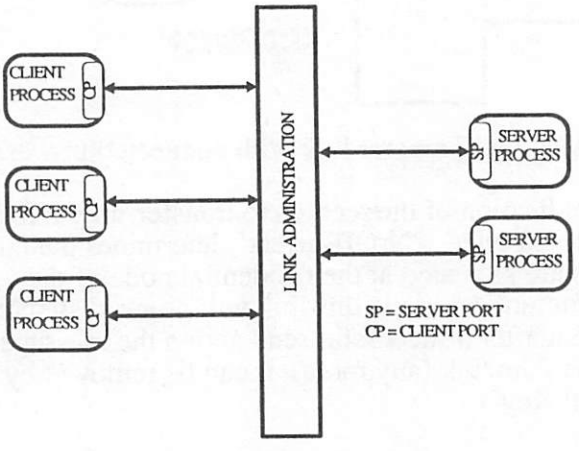


Fig. 4. A Remote procedure link with connected processes.

A RP link is created by any process by:

CreateRPLink ('LinkKey', <request type>, <reply type>);
If no process is connected to the link (any more), it can be removed by its creator by:
RemoveRPLink ('LinkKey')
Client - and server ports are declared in the following way respectively:
VAR cp: ClientPort;
VAR sp: ServerPort;
A client as well a server may connect to or disconnect from a link by:
Connect (xp, 'LinkKey');
Disconnect (xp);
in which "xp" is either a client - or server port.
A request "f" is sent and a reply is received in variable "a" as an *indivisible* action by the following procedure:
RequestReply (cp, f, a);
in which "cp" is a client port, "f" a value of <request type>, and "a" a variable of <reply type> in which the answer is received.
At the server side, the receiving of the request and the sending of the reply are separated in order to permit any form of programmable scheduling of the requested service: a request can be queued in order to postpone the reply. Therefore a unique identification of the sender process is received in the variable "sId" simultaneously with the request itself by:
ReceiveReq (rp, r, sId)
in which "rp" is a reply port, "r" is a variable of <request type>, and "sId" is a variable of the pre-defined type "SenderIdType".
A reply "g" is returned by the generic procedure:
SendReply (sId, g)
in which "sId" is a variable of the pre-defined type "SenderIdType" that contains a unique send identification and "rep" is a variable that contains the reply.

The ports connected to a RPC-link can be disconnected by using the procedure already introduced in section 4.2:
Disconnect (xp);
in which "xp" is either a client - or a server port.

RP links are easy to use by establishing distributed control structures. In particular distributed synchronization (distributed semaphores [Dijkstra 68], readers and writers problem [Courtois 71]) is easily established.
MT links are mainly used for communication structures in which no bi-directional link communication is needed. An important category is the sequential data transfer. In order to prevent for the merging of data streams, sender - and receiver access rights have to be controlled. This can be done by using a RP link as a control structure for a MT link. (In Tumult-16 we did not use an RP link but we introduced a "ClaimAsSender" and a "ClaimAsReceiver" primitive by which exactly one sender and one receiver could claim a MT link for sequential data transfer.)

5. Conclusion

In this paper a survey of Tumult-64 is given in which architecture and the communication aspects are highlighted.
The system is flexible in the sense that its hardware is modular extendible with respect to the number of used node processors. The system is dynamic in the sense that processes can be created dynamically and that they can be connected to, - or deleted from inter-process communication topologies (links) dynamically.
The introduced communication primitives, which, to our experience are powerful and easy to use, are believed to contain new aspects. Attention is paid to reliability, real-time behaviour and especially to the communication performance, which is because of the inherent parallelism and the speed of the network competitive. Up to 64 inter-process communications can be active at

the same time, provided that not more than 20 Mbytes per second (worst case, 40 Mbytes per second typical) is offered in total. Reliability of the data (record) communication is obtained by destroying mutilated messages after which underflow is detected and resending will requested. For control messages an alternating bit protocol is used. The network offers real-time facilities by guaranteeing a bandwidth per processor (With N processors this is 20/N MBytes/sec). The protocol overhead for *each* inter-process message transfer is expected to be in the order of magnitude of some 100 micro seconds, thanks to the dedicated protocol-handling hardware. Tumult-16 which is an ancestor and also a prototype of Tumult-64 is operational as a high performance prototype system for commercial applications of the Dutch PTT, in particular for handwritten character recognition.

Acknowledgements
We gratefully acknowledge the staff members of the University of Twente, the Dr. Neher Laboratory and Oce Netherlands R&D as well as the numerous students without whose support the implementation of Tumult would never have been possible.

References

[Artsy 87] : Artsy Y., Chang H.Y., Finkel R.: "Interprocess Communication in Charlotte", IEEE Software 4:1, Jan. 1987.

[Birrel 84] : Birrel A.D., Nelson B.J.: "Implementing Remote Procedure Calls", ACM Trans. on Comp. Syst., pp. 39-59, Feb. 1984.

[Black 85] : Black A.P.: "Supporting Distributed Applications: Experience with Eden", Proceedings of the 10th ACM Symposium on Operating Systems Principles, pp. 181-193, Dec. 1985.

[Bron 82] : Bron C.W.:"Modular Pascal Definition", Department of Computer Science, Twente University of Technology, The Netherlands, int. rep. nr. INF-82-10,1982.

[Bron 84] : Bron C.W., Dijkstra E.J.: "On the use of exception handling in Modular Pascal", Department of Computer Science, State University Groningen, The Netherlands, Sept. 1984.

[Cheriton 85] : Cheriton D.R., W. Zwaenepoel: "Distributed Process Groups in the V kernel", ACM trans. on Comp. Syst., pp. 77-107, May 1985.

[Courtois] : Courtois P.J., Heymans F. and Parnas D.L.: "Concurrent control with 'readers' and 'writers'" Comm. ACM 14, 10, pp. 667-68, oct. 1971.

[Dijkstra 68] : Dijkstra E.W.: "Co-operating Sequential Processes", Technical Report EWD-123, Technological University Eindhoven, The Netherlands (1965); reprinted in: Genuys F.: " Programming Languages", (Academic Press, London, 1968), 43-112.

[Feng 81] : Feng T.: "A survey of interconnection networks", Computer, Dec. 1981.

[Feng 84] : Feng T., Wu C.: " Tutorial: Interconnection networks for parallel and distributed processing", IEEE Computer Society Press, 1984.

[Hoare 78] : Hoare C.A.R.: "Communicating Sequential Processes", Commun. ACM, vol.21, no. 8, pp. 666-667, Aug. 1978.

[Hockney 85] : Hockney R.W.: "MIMD computing in the USA - 1984", Parallel Computing 2, pp. 119-136, 1985.

[Jansen 80] : Jansen P.G., Kessels J.L.W.: "The DIMOND: A component for modular construction of switching networks", IEEE Transaction on Computers, vol. C-29, no. 10, pg 884-889, Oct. 1980.

[Jansen 85] : Jansen P.G.: "Tumult, a distributed real-time multi-processor system", Proceedings NGI-SION 1985, (North Holland, Amsterdam, 1985), 196-208.

[Kessels 81] : Kessels J.L.W.: "The SOMA: A programming construct for Distributed Processing", IEEE Trans. on Softw. Eng. no. 5, Sept. 1981.

[Langen 87] : Langen P. v., Sijbers A.: "The distributed file system in Tumult", Proceedings Conference Computing Science in the Netherlands (North Holland, Amsterdam, 1987).

[Luttmer 89] : Luttmer M.L.M., Ribbers H., Jansen P.G.: "TUMULT-X: a real-time executive", University of Twente, Department of Computer Science, int. rep. nr. INF-89-26, 1989, ISSN 0923-1714.

[Miller 87] : Miller B.P., Presotto D.L., Powell M.L.: "DEMOS/MP: The Development of a Distributed Operating System", Software Practice and Experience 17, pp. 277-290, April 1987.

[Mullender 86] : Mullender S.J., Tanenbaum, A.S. "The design of a capability based distributed operating system", Computer Journal, Vol. 29, No.3 (1986)

[Needham 81] : Needham R.M., Herbert A.J., "The Cambridge Distributed Computing System", (Addison-Wesley, Reading, Mass., 1982).

[Rachid 81] : Rachid R.F., Robertson G.G.: "Accent: A Communication Oriented Operating System Kernel", Proceedings 8th ACM symposium on Operating System Principles, pp 14-16, Dec. 1981.

[Ribbers 87] : Ribbers H., Jansen P.G.: "Communication facilities of Tumult, a tutorial" Dept. of Computer Science, Twente University, The Netherlands, mem. inf-87-10, Feb. 1987.

[Rieger 80] : Rieger C., Bane J., Trigg R.: "ZMOB: a highly Parallel Microprocessor, Department of Computer Science, University of Maryland, Technical Report 911, 1980.

[Scott 87] : Scott M.L., Cox A.L.: "An Empirical Study of Message-Passing Overhead", 7th Conference on Distributed Computing Systems, Berlin, p.p. 536-543, September 1987.

[Scholten 87] : Scholten J., Smit G.J.M., Hofstede J.: "A proposal for an architecture for Tumult based on a serial data link", Proceedings Euromicro 87, Porthmouth: "Microcomputers: usage, methods and structures:, North -Holland Science Publishers, 1987, pp. 613-620

[Tanenbaum 85] : Tanenbaum A.S., Renesse R. v.: "Distributed Operating Systems", Computing Surveys, Vol. 17, no. 4, Dec. 1985.

[VME 85] : VME-bus specification manual, Revision C, Feb. 1985.

[Waumans 82] : Waumans B.L.A.: "Software aspects of the PHIDIAS system", Philips Technical Review, vol.40, nr. 8/9, 1982.

[Wirth 85] : Wirth N.,: "Programming in Modula-2", Third corrected edition, Springer Verlag, 1985.

Experiences with a Family of Multiprocessor Real-Time Operating Systems

Prabha Gopinath *
Philips Laboratories
North American Philips Corporation
Briarcliff Manor, NY 10510
psg@philabs.philips.com

Thomas Bihari
Adaptive Machine Technologies
Columbus, OH 43212
AMT@eagle.eng.ohio-state.edu

August 3, 1989

Abstract

We discuss our experiences with a family of operating systems designed for use in high-performance real-time applications. These experiences have been gathered over a period of six years. The paper discusses motivations, design tradeoffs, and practical issues that influenced the design and implementation of a family of operating systems beginning with GEM, and continuing through CHAOS and CHAOS-ART. All systems were designed for embedded real-time systems, and as such, include constructs for dealing with timing constraints.

1 Introduction

1.1 Motivation

Real-time software and the real world are intimately related. The effect of incorrect or unreliable software on the real world can range from the merely inconvenient to the disastrous. As a result, there is considerable emphasis on improving the correctness and reliability of real-time software, so that the frequency and consequences of failures are reduced. A proper methodology of programming real-time software can contribute significantly to the robustness and performance of the end-product.

The systems described here reflect the evolution in our thinking on how real-time systems should be programmed and what support the operating system should provide. We specifically target embedded real-time applications, including existing applications such as the Adaptive Suspension Vehicle (ASV) [2], and proposed applications such as the AMRF facility [3] and the NASREM space station [1].

Real-time software has several unique characteristics [4,5]:

- There is the need for enhanced reliability dictated the by potentially damaging consequences of failures.

- The sequencing and the timing of inputs to the system are determined by the real world and not by the programmer. Thus, a real-time application can expect to have conflicting demands made upon it, and should be able to deal with unexpected external events, or at least, *recover gracefully*. This means that as the demands on the application from external sources become more stringent, the behavior of the system will be altered. To meet our goals of predictable behavior, all such alterations in the behavior should follow predetermined patterns.

- Demands on the system typically occur in parallel. Hence, a real-time system generally has a parallel processing pattern, with either true or virtual concurrency. Support for synchronization at the system level thus becomes an important issue.

- Real-time programs must meet deadlines in order to satisfy the physical timing requirements of the real world. A "correct" real-time program is both functionally and temporally correct.

- Such systems typically have extended mission times, and so in addition to handling ordinary situations correctly, should also be able to recover from extraordinary ones.

In the domain of embedded real-time applications, it is customary to use custom operating system facilities that are optimized for the hardware and application environment. However, requirements for embedded systems often change during the lifetime of the product. Such changes may be the result of changes in the environment of the product, changes in the performance desired of the product, or perhaps changes that are made to other component parts of the product. With the complexity that we have encountered in our applications and with the significantly higher complexity that is inevitable in future systems, we feel that it is impractical to redesign operating system facilities in response to such changes. Rather the system software must be easily adaptable to the new set of requirements. Such adaptations may be dynamic - during a mission, or static - prior to a mission or between missions.

1.2 Real-Time Operating Software

Here we introduce the notion of *operating software* as application software that consists of application code and operating system software written and optimized in conjunction with each other. Several research problems exist dealing with the operating system utilities used by operating software:

- The task hierarchies and inhomogeneities in operating software imply that such software exhibits multiple grains of parallelism. Therefore, the operating system must support parallel application tasks of differing *weights*, ranging from small tasks that consist of a few instructions and that can execute at high frequencies with low overhead, to large tasks that execute infrequently. All tasks must be schedulable periodically or sporadically, and they must be scheduled, synchronized, and executed with overheads corresponding to their weights and within the strict time constraints determined by the application.

- As with task execution, communication requirements among tasks differ, therefore the operating system's communication mechanisms must efficiently implement a range of communication latencies. Furthermore, tasks may make different assumptions regarding the model of communication used [15]. For example, some tasks may tolerate the

loss of individual readings from a sensor in order to perform an operation at the highest rate possible, whereas other tasks may assume that individual messages never get lost. As a result, the operating system must support multiple models of task communication.

- Dynamic reconfiguration for increased reliability and performance requires that the operating system itself be reconfigurable, that it provide mechanisms to the application software for reconfiguration, and that information about the software's application and execution environment be available during program execution.

The use of the following principles of software development should result in software satisfying the functionality and performance requirements listed above:

1. Minimal "hardwired" runtime functionality.
2. Sharing of software components by selection and parameterization.
3. Modularity and support for adaptation.
4. Module synthesis to extend functionality and performance.

Minimal "hardwired" runtime functionality is attained if the number of software components that must exist on each computing node, (regardless of their use within the real-time application) is small. Thus, with regard to the operating system, only its "kernel" should exist on each node.

Selection and parameterization implies that if at all possible, the total number of software components in the system should be minimized by making them general-purpose, and shareable in nature. However, the unique characteristics of real time software, described in an earlier section, also constrain software components to have high performance. These conflicting requirements may be met by making software components adaptable, so that they may then be tuned on a use-specific basis. Such tuning of software can be done in two ways:

- By static or dynamic *selection* of software components - Rather than offering a single set of constructs for any one activity (such as process communication), operating software and its operating system components should offer a diversity of constructs of similar functionality with differing performance and reliability. Thus, operating software can be tuned (adapted) by its static use of a specific construct and by the dynamic substitution of one construct for another.
- By static or dynamic *parameterization* - Since some costs in memory space and execution time are associated with the selection and inclusion into software of alternative constructs, adaptations of software that concern performance and reliability are also realized by the parameterization of specific constructs.

Modularity and support for adaptation implies that the adaptation of operating software by selection and parameterization is possible only if modular, adaptable, and reusable[10] software components can be identified, and their interactions clearly defined.

Module synthesis means that once modules have been identified and their interactions clearly defined then the system should provide mechanisms by which appropriate compositions of these modules can be generated and used.

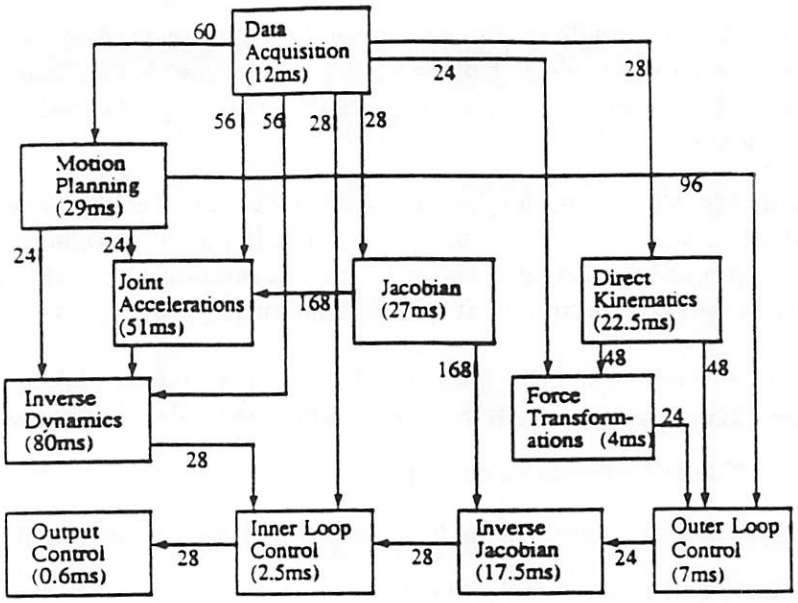


Figure 1: Robot Control Program

1.3 Typical Real-Time Systems

Figure 1 shows a typical real-time control application. Although the domain of real-time programs is extremely rich, this particular program exhibits characteristics common to many real-time programs as described later in this section (see also Section 1.1). The figure within each of the boxes is the execution time of the component, while the figure on each interconnecting link is the size of the interaction packet.

The software takes, as input, the current positions and velocities of a robot manipulator's joints and the desired positions and velocities of those joints. It then computes the necessary torque commands to be sent to the joint actuators, so that the joints may attain the desired positions and velocities.

The software consists of a number of components, some of which may run concurrently and some of which must run sequentially. These components have different execution times and the interaction packets that are exchanged among components are of different sizes. The allowable time between the arrival of a new set of inputs and the generation of a new set of torque values is bounded by a deadline that is determined by the physical properties of the robot manipulator. This is the *overall deadline* for the application. If this deadline is missed, then at best, the manipulator may fail to interact with the physical world in a timely fashion, and at worst, the manipulator may become unstable, thus endangering its operator and/or other workers.

To ensure that overall deadlines are met, "intermediate" deadlines are computed and assigned to the various components of the application. The scheduler for the application considers these intermediate deadlines as being 'sub-goals' that are sufficient conditions for the overall deadline to be met.

The application software is usually developed on a software development system associated with the target multiprocessor. The application programmer develops the application code using tools that are commonly available on such machines. These tools include, but are not restricted to, standard compilers, linkers, and loaders. Once a load module for the application has been generated, it is loaded into the multiprocessor and its performance is

measured. These measurements may necessitate redesign and remeasurement. The redesign could consist of changes in the code of various modules, changes to the module-processor map, and replacement of current versions of code with other versions that may be better able to meet the intermediate goal deadlines that have been statically computed.

Most real-time applications typically pass through several such modification cycles. Ultimately, the software is 'released' and becomes operational. During the operational stage of the software's life cycle, it may be continuously monitored and modified in a manner similar to that described above. Modifications made during this phase are driven by changes in the environs of the application that affect its performance and/or reliability. These modifications may be implemented automatically by software that has been designed-in for this purpose, or by systems maintenance personnel.

Changes, such as those described above, may be performed at different stages of an application's life-cycle. They may be major changes that require the interruption of the application and the use of development system tools, or they may be minor incremental changes that can be performed while the application is executing. Regardless of the magnitude and locale of these changes, the primary intent of all such modifications is to ensure that the software continues to meet its specified performance and reliability goals, usually in response to changes in its operating conditions.

We specifically address real-time applications, such as those just described, and describe systems that simplify the task of generating, adapting, and experimenting with such software. We also propose methods of generating application software that exhibits some degree of autonomous corrective behavior with regard to external events that may degrade performance and reliability to unacceptable levels.

1.4 Outline

The remainder of the paper discusses the GEM, CHAOS, and CHAOS-ART operating systems. Figure 2 illustrates the design history of these systems.

In each case we first present a brief overview of the particular system. Next we discuss some of the performance and adaptation aspects of the design. Finally we critique the design and motivate some improvements. We will continually refer to the four principles of software development outlined earlier.

2 The GEM Operating System

2.1 Overview

The GEM (Generalized Executive for Multiprocessors) operating system was specifically constructed for a large robotics application, namely, the operating software executing on the embedded multiprocessor of the prototype of a six-legged mobile vehicle, the Adaptive Suspension Vehicle (ASV) [18]. The main attributes of GEM [15] are:

- It supports two different sizes of tasks called processes and micro-processes[1], and it offers a variety of scheduling calls.
- Process and micro-process switching are performed within the real-time constraints acceptable to relatively high-speed control tasks.

[1]Though micro-processes were implemented after the ASV software had been completed.

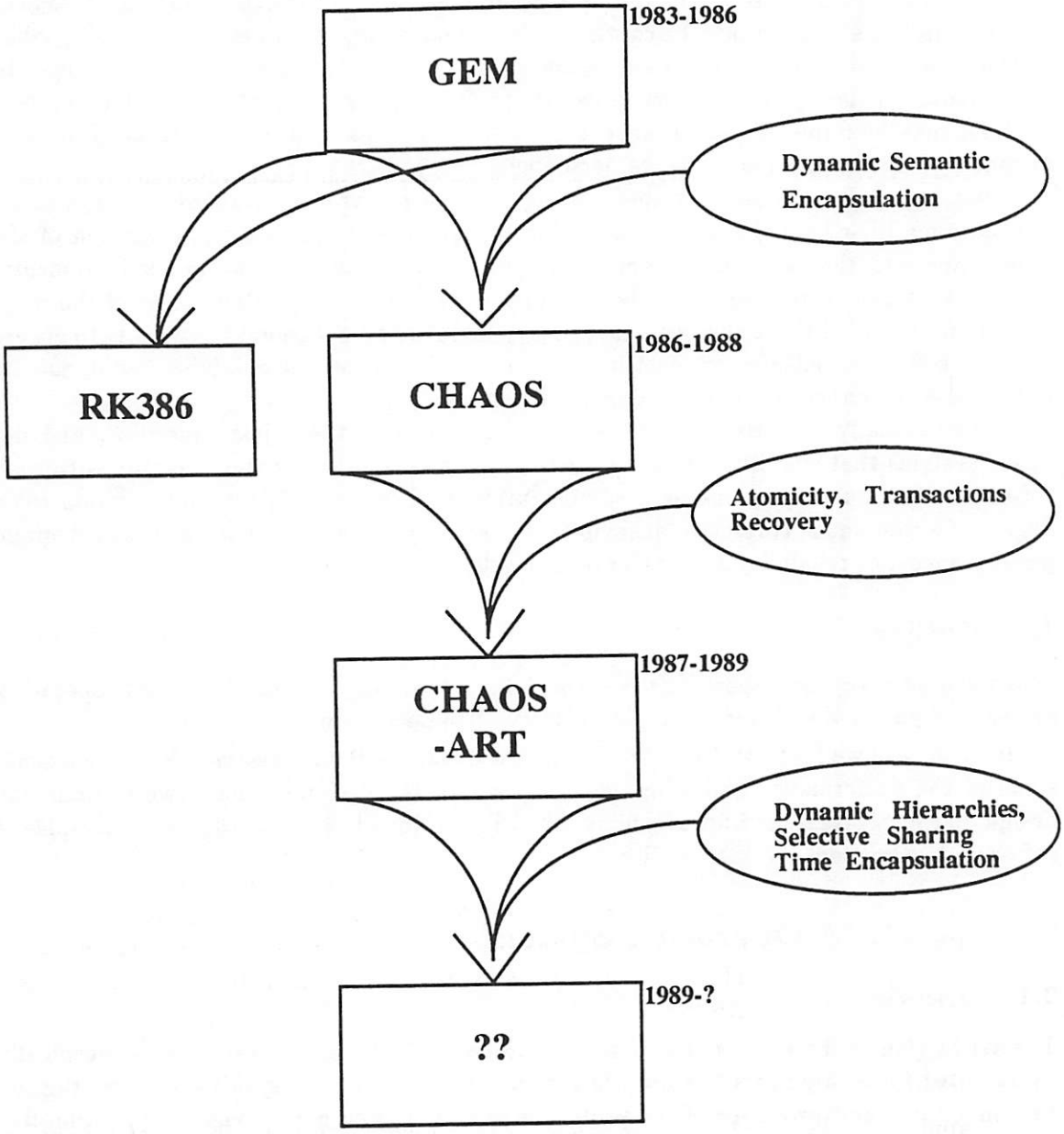


Figure 2: The GEM, CHAOS, and CHAOS-ART Family

- It supports multiple models of communication with a mechanism that is parameterized with respect to communication bandwidths and speed.
- It supports a number of static adaptations of operating software, including alterations of scheduling and communication mechanism parameters.

The development of GEM constituted our first attempt to apply the first three principles regarding high-performance operating software defined in Section 1.2:

1. Minimal hardwired functionality - The operating system nucleus for each node of the embedded system is quite small (roughly 18 Kbytes code and data in the ASV version of GEM). All other operating system functions exist within each node only if needed. Operating system functions not required on each node locally are instantiated as user-level processes on any one node, and are replicated only when required for reliability or performance .

2. Sharing by selection and parameterization - GEM provides a simple generic model of tasking. GEM users can select one of two task sizes, processes or micro-processes, which differ regarding their latencies of activation in response to requests and in that the micro-processes contained within a single process cannot execute in parallel. Processes are used to represent independently schedulable and potentially parallel real-time tasks, whereas micro-process serve to structure the different activities performed within a single real-time task in response to the variety of stimuli handled by the task. GEM supports a simple mailbox-based model of communication. Users can realize different flavors of the base message-passing primitive by appropriate parameterization of the communication primitives. Thus, different functionality, performance and reliability can be obtained.

3. Modularity and support for adaptation - All operating software is organized as processes and micro-processes, that interact by the use of shared memory and GEM's communication primitives. Some programming system support exists for the static selection of task sizes and for customizing the message-send operations. Several operating system constructs concern the dynamic adaptation of specific parameters regarding tasks and task communication. In addition, GEM allows copies of a process to be statically loaded onto multiple nodes. When load balancing requirements exist, a process on one node can be forced into a quiescent state, its state transferred to a process on another node, and the target activated. This can also be done to address reliability concerns [2,16].

2.2 The ASV Application

GEM's application software and computer hardware exhibit the attributes of operating software listed in Section 1.2 (see Figure 3 and the following text). The ASV vehicle is operated with high-level commands issued by a human operator. These commands are translated into vehicle actions by operating software structured as a hierarchy of processes performing higher vs. lower level control tasks (e.g. see BMP (Body Motion Planning) vs. BS (Body Servo) and LMP (Leg Motion Planning) vs. LS (Leg Servo) below). Additional inputs to the system's operating software are provided by an inertial guidance system on the body, by pressure, velocity, and position sensors, and by an optical terrain scanner.

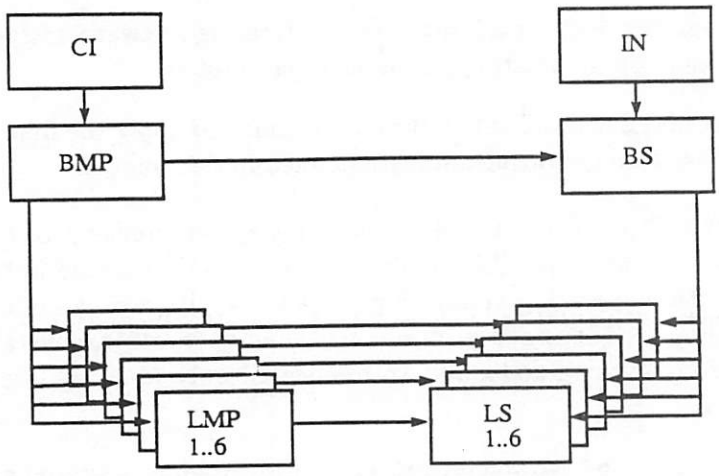


Figure 3: The ASV Robot's Operating Software

The robot's operating software consists of multiple tasks executing either asynchronously or periodically at varying rates. The following informal descriptions of some tasks and processes specify task inputs, outputs, and execution rates (see [18] for a more detailed description of the ASV's software):

- (CI) Cockpit Input and Cockpit Display - GEM processes that respectively accept commands from the operator at rates determined by human bandwidth and communicate with the display device at several Hz; CD displays information on a periodic basis or on demand; CI formats commands appropriately, and passes them to (BMP);
- (BMP) Body Motion Planning - A GEM process running at a rate determined by possible rates of change in movement of the vehicle's body (body control bandwidth), about 20 Hz. It takes high-level commands from (CI), modifies them to ensure stability of the body, determines the necessary rates of leg movement and sends them to (LMP); also sends commands for the desired body position, velocity, and acceleration to (BS);
- (LMP) Leg Motion Planning - One task per leg and one GEM process per task, each process running at about 50 Hz; A task determines the actual leg trajectories while they are in the "transfer phase" (i.e., in the air) and sends commands to (LS);
- (BS) Body Servo - A GEM process running at a rate determined by the body control bandwidth, about 20 Hz; takes information from (IN) and (BMP), calculates the necessary actuator pressures when the legs are in the "support phase" (i.e. on the ground), and sends them to the (LS) processes;
- (LS) Leg Servo - One GEM process per task, each running at a rate determined by the bandwidth of the hydraulic system, about 100 Hz; sends the actual commands to the hydraulic system;
- (IN) Inertial Navigation - These GEM processes, running at about 20 Hz and 5 Hz, process sensor information and send it to (BS);

- (TMG and G) Terrain Map Generation - (not shown in figure) These GEM processes process data generated by the terrain scanner used by the vehicle guidance (G) system, which consists of 4 GEM processes, each running at about 1 Hz, controlling body velocities and selecting footholds when the vehicle is in terrain-following mode.

Most processes are structured as several internal, or separable activities. In addition to the periodic tasks above, several *sporadic* tasks, including (DL) Data Logging and (I) Initialization, support computation, collect test data for analysis, and handle errors.

Figure 3 displays most tasks of the robot's operating software as well as their communication and control relationships. The attributes of real-time software listed in Section 1.2 can be identified:

First, several constructs in GEM, believed to be specifically useful in the real-time domain, suited the requirements of the ASV application.

Second, the control software is structured hierarchically, where processes performing higher level control, e.g., (BMP), loosely interact with processes performing lower level control, e.g., (BS) and (LMP). Furthermore, the entire application is not simply described in terms of replicated components. Instead, its component processes differ widely in execution rates and execution times.

Third, process interactions are of three different kinds (not differentiated graphically). They include (a) unqueued (and thus potentially prone to loss), very low latency data communications between (LS) and (LMP), which simply wish to share the most recent data values they generate, (b) queued (and therefore not susceptible to loss) control communications between (CI) and (BMP) with which the vehicle operator issues commands to the vehicle's control software, and (c) data communications that are queued only up to a certain age for the sporadic processes logging recent system events.

Fourth, dynamic reconfiguration, although not supported by GEM structures, is necessary when the vehicle's operator changes operating modes (e.g., from slow, precise foot placement to fast vehicle movement along even terrain) or when certain exceptional conditions occur (e.g., encountering unexpected soft terrain).

2.3 Lessons From GEM

We now discuss why GEM's operating system primitives, and application software based on GEM's process-based paradigm, simply do not allow programmers to represent sufficient information about the parallel structure and the semantics of application software to facilitate its static and dynamic reconfiguration. This is of special importance given our requirements of adaptability, efficiency, predictability.

- GEM supports functional decomposition of application software. For reasons of efficient access, shared data structures are global. Any changes to such data requires that the implementation of most, if not all, of the functional modules be changed.

- Adaptability, and especially dynamic adaptability where the software is being modified at run-time, requires that all semantic information about the application specified during program construction be available at runtime.

- Efficiency of execution requires that all such information required for adaptation be appropriately grouped and maintained by the operating system. This differs from the programming environment approach where semantic structure is preserved only during

the static phase. At run time however all such semantic information is lost. Absence of semantics is exemplified by the use of GEM's communication channels for both control and data interactions among tasks. Absence of constructs for the representation of parallel structure is exemplified by the fact that the operating system cannot represent functional groupings of processes[7,11]. Since the application can be abstracted only to a limited extent, on-line control and adaptation of well-defined parts of the application is difficult. The operating system does not support the programmer in defining such parts, and as a result difficulties arise in software debugging and maintenance.

While GEM has been used successfully for the current ASV software, some important lessons have been learned regarding the manner in which to apply the three principles of operating software development listed in Section 1.2. Regarding minimal hardwired functionality, GEM is quite successful in that its nucleus is small. However, regarding the adaptation of operating software, several items should be noted:

1. Adaptation by selection - GEM should offer more than two task sizes (processes and micro-processes), and facilities that allow the application to be structured at different levels of abstraction, so that "larger" units in the hierarchical structure of the ASV software can be represented. For example, the six, replicated task pairs LS-LMP should be represented as one functional unit, thereby not reducing the number of processes, but reducing the complexity of on line software control, and of the description of the ASV's operating software. We note that the number of GEM processes and communication channels in the ASV's software currently exceeds one hundred.

2. Adaptation by parameterization - The implementation of GEM's process-to-process communication construct is such that there is very little performance gained by reducing the susceptibility of messages to loss. This is due to the low-level nature of that construct and certain peculiarities of the address mapping hardware on the 86/30 processor nodes on which GEM runs.

3. Modularity and support for adaptation. - During execution, processes and micro-processes interact by exchange of messages via shared communication channels. Since all communications, regardless of their nature (e.g. control messages vs. data messages) are sent in this fashion, and are therefore indistinguishable from each other at runtime, the dynamic reconfiguration of GEM's operating software is difficult (with the exception of process migration, of course). For example, the replacement of a communication link by a different link (e.g. one that is more reliable) is not easily performed, since nothing is known about the manner in which the link is used.

Hence the process and mailbox model of GEM is not the proper level at which an application programmer should design and represent a complex, parallel, real-time application where adaptability, efficiency, and predictability are important criteria.

2.4 Summary

GEM has been ported to a Multibus II, 80386-based shared-memory multiprocessor. Extensions include the provision for small protected address spaces. In fact each buffer in a GEM mailbox, an *Envelope* in GEM parlance, is a protected address space. A GEM process

can consist of several such address spaces, each protected from the other. This version of GEM, called RK386, is being used for current versions of the ASV application.

This section described GEM, a process and mailbox-based, operating system. It described how GEM was designed to adhere to the basic principles for development of real time operating software that were set forth in Section 1.2. A large real-time robotics application was used to provide examples of GEM's features and shortcomings primarily in light of the goals of efficiency and predictability that we had set forth earlier. The lessons learned from the GEM experience were described. In the next section we describe the CHAOS system and show how it has also been designed using the principles defined in Section 1.2.

3 The CHAOS System

3.1 Overview

CHAOS - A Concurrent Hierarchical Adaptable Object System, is a complete programming and operating system for embedded real-time software. Its goal is to support the programming of real-time applications that are efficient and predictable.

To achieve these goals CHAOS incorporates the following components (see Figure 4):

1. The CHAOS object based programming and execution model, which allows application programmers to describe applications in terms of concurrent objects interacting with invocations. The notion of objects is supported through every level of the CHAOS system, including the operating system.

2. An Entity/Relationship (**E-R**) data representation framework augmented with Action Routines. The E-R database represents functional and performance attributes of the application and contains complete descriptions of compile and runtime representations of all objects and their interactions.

3. An Adaptation Control System(ACS) consisting of:

   - A Monitoring system(MON), which observes and reports runtime information through embedded sensors.
   - A Data Management System(DMS) which stores information about the application. The DMS uses the E-R framework described above.
   - An Adaptation Controller(AC) which decides what adaptations must be made and when such adaptations should be made.
   - An Adaptation Enacter(AE) which actually performs the adaptations selected by the AC.

4. The CHAOS run-time system which controls the execution of objects, handles the delivery and scheduling of invocations among objects, and maintains information about objects and their attributes. Such information maintained at run-time allows for efficient dynamic adaptation of executing software.

5. COLD, a declarative language accessed through a syntax-directed editor, that provides a high-level interface to the CHAOS system. COLD constructs describe object structures, object interaction patterns, and application timing characteristics and

constraints. Executing a COLD program generates, among other things, the E-R representations in the DMS.

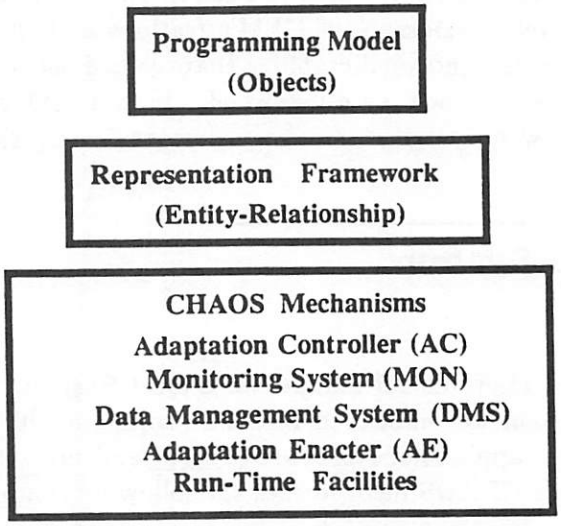


Figure 4: CHAOS - System Components

3.2 CHAOS - Objects

In the CHAOS implementation of the object model, a parallel program is described as a set of abstract *objects* that interact by invocation of each others' operations. Each object has a type, unique name, and list of operations. Object types are user-defined and are not dynamically checked or known to the operating system. To invoke its operations, only the identity of the object and its operations' names need be known, so that the parallel operating software generated from a set of objects specified by the programmer may be adapted considerably without changing its object description. For example, an object can be a *passive* object - its operations and data are implemented by code modules executed within the thread of execution of the invoking object - or it can be an *active* object - its operations are realized either as a single server process or as a set of executable server processes.

The actual number of processes associated with each object is determined by performance or reliability considerations (e.g., the frequency of invocations on the object). Multi-process objects are controlled by a single coordinator process accepting object invocations and scheduling them for processing by one of several server processes. The structure of a typical CHAOS object is depicted in Figure 5.

3.3 CHAOS - Invocations

In CHAOS, the act of requesting that an operation be executed is termed an *invocation.* CHAOS provides a rich set of invocation primitives that have been tailored towards the performance and functionality requirements of real-time applications. CHAOS supports three distinct classes of invocations, a purely control invocation where there is no transfer

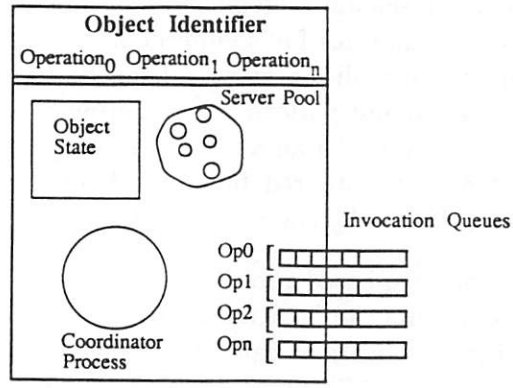


Figure 5: A CHAOS Object

of data(ObjFastInvoke), a purely data invocation which although it does require control transfer for set-up, degenerates to data transfer(ObjStreamInvoke), and an invocation that provides a combination of the two(ObjInvoke).

An important feature of a CHAOS invocation is that it executes in two distinct phases, separating control transfer from data transfer. The first phase of every invocation involves a transfer of control information from the invoker to the invokee. The initial stage of the control phase executes in the invoker's context. As part of the control phase of the invocation, the request is enqueued (if appropriate) on the target object. The remainder of the control phase executes in the invokee's thread of control. The coordinator process at the invokee object examines its invocation queues, selects a request to service, dequeues it and schedules a server process to execute the requested operation. The data transfer phase occurs in the context of the server process. Thus, from the point of view of the invoker object, invocations are lightweight, since a substantial portion of the cost of performing the invocation, including data transfer, is incurred by the target of the invocation. Invocation requests are entered into a Deadline queue, where a request's position is determined by a deadline value associated with the invocation.

3.4 CHAOS - Specializations

CHAOS demonstrates that the object oriented model of software can be specialized and implemented efficiently so that it can be applied in the real-time domain. The following specializations and implementation attributes exist:

- Objects of different "weights" may be created, ranging from light-weight, passive objects that have no internal processes, to heavy-weight objects that may have multiple internal processes. Therefore, in contrast to the micro-processes within a GEM process that cannot execute concurrently, an object may exhibit internal parallelism. For example, for the ASV application, the replicated, concurrently executing LS-LMP pairs may be grouped into a single CHAOS object.

- In order to implement efficient object interactions, invocations of an object's operations can have different semantics, performance, and reliabilities, ranging from (a) invocations that entail the transfer of control and parameter-passing (much like RPC implementations [5]) through (b) streaming invocations with low incremental cost of data transfer to (c) extremely fast control invocations that may be used to toggle

actions. This multiplicity of semantics demonstrates that existing formulations of the object model or of RPC semantics for computer networks are not trivially applied to the real-time domain. In addition since a fixed library of invocation primitives may not meet the functional and performance requirements of every real-time applications, mechanisms exist by which an application programmer can *synthesize* invocation primitives that have the desired functional and performance characteristics. Invocation synthesis is achieved by composition of a set of basic building-blocks.

- In contrast to GEM's message-based process interactions, explicit scheduling parameters and real-time constraints can be attached to object invocations. The scheduling policy for servicing object invocations can be controlled and changed as well.
- Since objects can reside anywhere in the multiprocessor hardware, the invocation code can select the communication link to reach the target object that best fits the invocation's required performance or reliability. These links include the system bus, and serial and parallel links. In addition, CHAOS allows the programmer to explicitly control the visibility of objects by locating them in 'low latency of access' local memory or in 'high latency of access' dual-port memory.

3.5 CHAOS - A Critique

In summary CHAOS has been able to successfully address some of the more obvious problems we had identified in GEM. However, continuous experience with it suggested significant further improvements. Note that CHAOS has no support for atomicity and no provisions for recovering from failed invocations. While the adaptation mechanisms can be used to delay failures, by adjusting deadline schedules and by selecting versions; in an extremely "hostile" and rapidly changing environment, invocations can be expected to fail for various reasons. CHAOS lacks any support for recovering from failures.

4 The CHAOS-ART System

The design of CHAOS concentrated adaptations that anticipated changes in the operating environment - *Preventive Adaptations.* The design of CHAOS-ART encompasses adaptations that react to changes in the operating environment - *Reactive Adaptations.* CHAOS-ART is an extension to CHAOS that supports nested atomic actions as the basic mechanism for synchronization and recovery. Our experiences with the ASV, a robotics tracking application, and CHAOS, have shown that atomicity fits well with the concurrency control and recovery requirements of real-time systems. However, implementations for database applications or distributed computing [20] contradict a real-time system's requirements of concurrency, responsiveness, urgency, and high performance. Specifically, traditional definitions of concurrency atomicity assume the scheduling of concurrent activities to enforce a serialized order (e.g. Two-phase locking). This is undesirable in real-time systems because it reduces potential concurrency. As a result, the system will not always be able to respond to asynchronous events if the responding activity has to compete with other ongoing activities. This reduces the responsiveness of the system and may disturb execution scheduling required by the urgency hierarchy.

Similarly, regarding recovery of real-time actions, such recovery cannot always be achieved by rolling back to some previous "consistent" state because: (1) Time can not be rolled

back. The time spent in doing the (partial) action can't be "unspent". (2) If an action affects the external environment in any way, it might be impossible to undo that effect (some physical processes are irreversible, e.g. launching a rocket, moving a car in a one way street, etc.). (3) If the action responds to external events, undoing it implies either losing those events or regenerating them. Losing the events means that the action is not completely undone while regenerating them means that they were delayed as a side effect of the failed action. In both cases, the action is not really undone.

To compensate for the potential degradation in concurrency of strongly serialized atomic actions, CHAOS-ART relaxes the strict two-phase locking scheme allowing the programmer to release locks before the action terminates, provided that it would not result in an inconsistent state. Responsiveness to asynchronous events is improved by using *revocable locks*. Revoking a lock from an action can either abort the action (atomic) or delay it until the lock is returned (non-atomic). The system does not guarantee the consistency of shared resources with non-atomic locks. By revoking a lock, an "urgent" action can preempt less urgent ones competing for the same lock, thus enforcing the required hierarchy of urgency levels.

Since backward recovery of aborted actions is not always possible, CHAOS-ART supports both backward and forward recovery (Compensatable Objects [19]). Backward recovery is provided automatically by the system; the recoverable state is automatically restored when an action aborts. Forward recovery is provided by the programmer in the form of a *compensation operation* to be executed in case of abortion.

Pre-scheduling provides a "practical" compromise between the high overhead associated with atomic objects and the high performance often required by real-time systems. It also helps in absorbing transient overloads by anticipating future activities and preparing for them ahead of time. Pre-scheduling uses the slack time in most application executions to distribute the additional overhead imposed by atomic actions. Activities that can be pre-scheduled include: (1) Lock acquisition: the locks that are needed by an invocation may be acquired before the invocation actually starts. The definition of a lock has been extended to allow locks to be *pre-acquired*. A pre-acquired lock is kept by an activity as long as no other activities are waiting on it. If a running activity tries to acquire a pre-acquired lock, the lock is granted immediately. If another activity tries to pre-acquire it, the scheduler decides which activity can keep the lock. The decision depends on the scheduling policy being used. A reasonable policy is to grant the lock to the activity with the closest deadline. (2) Copying the recoverable state: a delayed invocation tries to keep a recent copy of the recoverable state. The object's scheduler maintains a list of all invocations with copies of the recoverable state. If it changes, the scheduler invalidates the copies and starts updating them. (3) Pre-scheduling invocations: invocations within the body of a delayed object operation are recursively pre-scheduled. Information about these invocations can be either provided by the programmer or automatically generated by the language processor.

Objects in CHAOS-ART can be either atomic or non-atomic. For *NonAtomic* objects, the system does not provide any support for the consistency of the object's state. It is the responsibility of the object's programmer to make sure that it can withstand arbitrary failures and to synchronize parallel invocations. Non-atomic invocations always succeed.

Atomic objects provide some degree of serialization and failure recovery. In general, atomic objects can be made fully serializable at the expense of potential concurrency by using strict two-phase locking. Each operation specifies a set of locks that are to be automatically acquired before starting the invocation. These locks are not released at the end of the invocation. Instead, the system keeps a list of these locks and automatically releases

them at the termination (success or failure) of the action doing the invocation. Full serialization can be achieved by using an exclusive lock and requiring each operation to acquire that lock at the beginning. This scheme, however, does not allow any concurrency at all. Knowing the semantics of the object, the programmer can define a combination of locking patterns that will allow more potential concurrency and still maintain the consistency of the object.

Two recovery schemes are provided for atomic objects: forward and backward recovery. **Forward** recovery is defined by the programmer by means of a recovery procedure for each recoverable action. The recovery procedure should semantically undo the effects of the action. **Backward** recovery is similar to the conventional recovery provided in data base systems and is achieved by splitting the state of each atomic object in two parts: **recoverable** and **non-recoverable** state. A single copy of the non-recoverable state is shared by all invocations which means that the changes made to it can't be automatically undone (they can be undone, however, by the forward recovery procedure). The recoverable state, on the other hand, is not shared by multiple invocations of the same object. Instead, each invocation manipulates a local copy of the recoverable state that is discarded if the invocation aborts. If the invocation commits, the local copy becomes the current object state and the old state is discarded. Full backward recoverability can be achieved by making the whole state recoverable.

Invocations: An atomic invocation can have a deadline, a delay, a start condition and a stop condition. Atomic invocations are implemented in four parts: The **Prologue**, **Body**, **AntiBody** and the **Epilogue**. The **Prologue** can be supplied by either the programmer or the system. It specifies the activities to be performed before entering the body. There are two kinds of Prologue activities: lock acquisition and specification of pre-scheduled invocations. The Prologue is separated from the Body for efficiency reasons; while the body is waiting to start, the Prologue can start acquiring locks and creating templates for future invocations. The **Body** is provided by the programmer. It is activated by the system after both the delay specified in the invocation expires and the start condition is enabled. The body can be interrupted if the stop condition is enabled before it commits (failure). The **Anti-Body** is an optional forward recovery procedure that is provided by the programmer. It is activated if the body fails. The anti-body has the same view of the object's state as the Body. The **Epilogue** is provided by the system. If the action fails (aborts), it is activated after the anti-body is executed. On the other hand, if the action succeeds (commits), it is activated after the top level action succeeds (commits). The Epilogue has the task of cleaning up after the action. For an aborted action, it releases all locks and aborts all siblings. For a committed action, it updates the recoverable state, commits its siblings, and releases all locks.

Transactions: A transaction is started by invoking an operation of an atomic object and terminated when the invocation terminates. As with atomic invocations, a transaction can either commit or abort. An aborted transaction should not have any 'semantic' effect on the global state of the system. Transactions can be nested by having a transaction invoke an atomic operation. A parent transaction does not terminate until all of its siblings have terminated. Failures of transactions automatically propagate down the tree (if a parent fails, all of its siblings fail) but not upwards. An atomic invocation can be designated as a **top level** transaction which has the effect of disconnecting this transaction from its parent and viewing it as being independent (i.e. can succeed or fail regardless of its creator).

Locks: Locks are used to synchronize concurrent accesses to shared resources. A lock is characterized by two attributes: exclusive/shared and atomic/_non-atomic. Non-atomic

locks are explicitly acquired and released by the program. When full serialization is required, atomic locks are used. The program acquires atomic locks as required but does not release them explicitly. Instead, atomic locks are automatically released at the end of the transaction. The **Revoke** operation is defined on locks, in order to allow critical activities to take away locks from less critical ones. Revoking an atomic lock aborts the transaction holding it while revoking a non-atomic lock merely delays the transaction until the lock is reacquired. In the latter case the programmer is responsible for the consistency of the system state. Revokable locks support pre-scheduling by allowing an activity to pre-acquire a lock that can be later revoked. They also provide a way of detecting failures; if an activity is using a device (e.g. robot arm) it is supposed to be holding a lock on that device. If the device fails, the lock is revoked and as a result, the activity will be affected (aborted if the lock was atomic and delayed until the device is fixed otherwise).

5 What Next?

In this section we outline our current thinking with regard to using CHAOS/_CHAOS-ART in embedded applications. There are three major issues: 1)Selective Sharing among objects, 2)Dynamic Hierarchies, 3)Temporal Encapsulation.

Selective Sharing: While, in general, the object and invocation paradigm of programming is suitable for a wide variety of applications, in the realm of real-time software the overheads incurred may, at times, be significant. This is especially so, when the operation that is being invoked is lightweight.

Invoker Operation	Invokee Operation	Round-Trip Time (msecs)	Operation Execution (msecs)	Invocation Overhead (msecs)
GrabBlock	Survey_Part	190	170	20
GrabBlock	Adjust_IKin	510	320	190
MoveJoints	Move_Robot	250	160	90
Move_Robot	ReadData	40	1	>30

Table 1: Measurements: Tracking Application

Table 1 represents some measurements from a robotics tracking application and shows that object operations have widely varying granularities. Thus for instance the Adjust_IKin operation has an execution time of approximately 320 msecs whereas the frequently invoked operation, Read_Data, has an execution time of approximately 1 msec. This illustrates the first shortcoming of our specialization of the object model. CHAOS rigidly enforces the boundaries among objects and requires that all interactions among objects take place only through the invocation mechanism. In some cases this results in overheads that are grossly larger than the cost of the operation itself. To put this in perspective, a procedure call on the specific 8086 based multiprocessor costs only 0.82 msecs. This suggests that in cases such as we have described, controlled sharing of address spaces among objects

should be permitted. On shared-memory machines such as the Butterfly, such sharing would be effected by allowing objects to selectively expose portions of their state to other objects in the system. Such selective compilation of one object into another object's address space allows invocations between them to be treated as procedure calls - thus avoiding the overhead of an invocation.

Dynamic Hierarchies: CHAOS/CHAOS-ART allows functional groupings of objects, but the run-time view of objects and groups of objects is flat. CHAOS does not support dynamic groupings of objects. In both systems objects are viewed and represented as self-contained entities that do not depend on other objects for any of their properties. The ability to abstract object functionality has been shown to be a powerful structuring tool in HPC [12] and RESAS [4]. It is acknowledged that hierarchical structuring leads to run-time inefficiency especially in cases where the level of hierarchical composition is high. For example in the subclassing form of inheritance of Smalltalk [8], when an object receives a message, resolving the method requires that its superclass chain be searched till the appropriate method is found. Techniques such as in-line caching, and static resolution of method names reduce this overhead, but are suitable only when dynamic adaptations are not required. Traversing a class hierarchy tree at run-time would cause severe performance penalties in a real-time system, and motivated the original decision that CHAOS objects would be flat. However certain specific applications stipulate that an object can be invoked only by the object directly above it. In fact an object is not even aware of the structure of the object directly below it. In effect a restriction such as this minimizes the run-time overhead of hierarchies since now lengthy chains of objects no longer have to be traversed. We feel that such constrained hierarchies have great value in the real-time domain. Powerful dynamic adaptation of the software is possible by substituting entire object hierarchies at run-time.

Temporal Encapsulation: A final comment we have relates to the lack of temporal encapsulation facilities in current object-based systems. The great advantage of the object model of software structuring lies in the *functional* encapsulation that it promotes. Thus the functional interface presented by an object is completely independent of the implementation of the object. In real-time systems, programs have to deal with the added dimension of temporal characteristics. In CHAOS, objects interact among themselves through invocations. All objects cooperate so as to meet some overall high-level goal, such as performing a high-level action subject to some specified high-level time constraint. The ability of the application to satisfy a timing constraint is dependent on the temporal behavior of the objects and on the characteristics of the invocation primitives that they use. As stated in Section 1.1, the success of complex dynamic applications depends on the ability of the constituent objects to adapt their execution [3,9] to suit prevalent conditions. The problem, however, is that while proper techniques of object-oriented design ensure that adaptations such as these do not affect an object's external *functional* interface, they offer no such assurance for an object's external *temporal* interface. Consequently, when an object's implementation is changed, often the temporal behavior of all objects that interact with it may have to be modified. The problem is therefore quite analogous to the situation in function-oriented design, described by Booch [6], where changes to global data necessitate changes to the implementation of many modules.

6 Acknowledgements

The research described in this document was sponsored by NSF grant ECS-8307216 and by the Defense Advanced Research Projects Agency (DOD) under contracts MDA903-82-K-0058, DAAE07-84-K-R001, and MDA972-88-C-0031. Several equipment grants from Intel supported the research into the CHAOS system.

GEM: The GEM operating system was developed by a team consisting of Karsten Schwan, Bruce Weide, Thomas Bihari, and Gregor Taulbee, all of the Department of Computer Science at the Ohio State University. Implementation was primarily by Thomas Bihari, with cooperation from members of the Electrical Engineering Department.

CHAOS: The CHAOS system was developed by a team consisting of Karsten Schwan, Prabha Gopinath, and Win Bo, all of the Department of Computer Science at the Ohio State University. Implementation was primarily by Prabha Gopinath with assistance from Thomas Bihari and Win Bo.

CHAOS-ART: The CHAOS-ART system was developed by Karsten Schwan and Ahmed Gheith while at the Ohio State University. They are continuing with this work at the Georgia Institute of Technology.

The work described in this document has led to four PhD theses, several masters theses and numerous short-term projects.

References

[1] J.S. Albus, H.G. McCain, and R. Lumia. NASA/NBS standard reference model for telerobot control system architecture (NASREM). Technical report, National Bureau of Standards Tech. Note 1235, July 1987.

[2] Karsten Schwan Thomas E. Bihari and Ben Blake. Adaptive, reliable software for distributed and parallel, real-ti me systems. In *Sixth Symposium on Reliability in Distributed Software, Willi amsburg, Virginia*, pages 32–44. IEEE, March 1987.

[3] Thomas Bihari and Karsten Schwan. A comparision of four adaptations for increasing the reliability of real-time software. In *Proceedings of the Ninth Real-Time Systems Symposium, Huntsville, AL*, pages 232–243. Assoc. Comput. Mach., 1988.

[4] Thomas E. Bihari. *Adapting Real-Time Software For Reliable Performance.* PhD thesis, The Ohio State University, 1987.

[5] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[6] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, Feb. 1986.

[7] David R. Cheriton and Willy Zwaenepol. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[8] Adele Goldberg et al. *Smalltalk-80. The Language And Its Implementation.* Addison-Wesley, 1984.

[9] Prabha Gopinath and Karsten Schwan. CHAOS: Why one cannot have only an operating system for real-time applications. *ACM Operating Systems Review*, C-36:to be published, In press 1989.

[10] T.C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE-10(5):488–493, Sept. 1984.

[11] Thomas J. LeBlanc and S.A. Friedberg. Hierarchical process composition in distributed operating systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado*, pages 26–34. IEEE, ACM, May 1985.

[12] T.J. LeBlanc and S.A. Friedberg. Hpc: A model of structure and change in distributed systems. *IEEE Transactions on Computers*, pages 1114–1129, December 1985.

[13] C.R. McLean and P.F. Brown. The automated manufacturing research facility at the national bureau of standards. In *Proceedings of the IFIP W.G. 5.7 Working Conference on New Technologies for Production Management Systems, Tokyo, Japan*, pages 240–249, Oct. 1986.

[14] W.J. Quirk. *Verification and Validation of Real-Time Software.* Springer-Verlag, 1985.

[15] Karsten Schwan, Thomas Bihari, Bruce Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[16] Karsten Schwan and Ben Blake. A fast scheduling mechanism for real-time systems. Technical report, Computer and Information Science, The Ohio State University , OSU-CISRC-5/87-TR16, Sept. 1987.

[17] John A. Stankovic. A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[18] Thomas M. Walliser Thomas Bihari and Mark R. Patterson. Controlling the adaptive suspension vehicle. *IEEE Computer Magazine*, 22(6):to be published, June 1989.

[19] H. Tokuda. Compensatable atomic objects in object-oriented operating sysytem s. In *Proc. of Pacific computer communication Symposium*, October 1985.

[20] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

Implementation Issues for the Psyche Multiprocessor Operating System

Michael L. Scott
Thomas J. LeBlanc
Brian D. Marsh

University of Rochester
Department of Computer Science
Rochester, NY 14627
{scott,leblanc,marsh}@cs.rochester.edu

ABSTRACT

Psyche is a parallel operating system under development at the University of Rochester. The Psyche user interface is designed to allow programs with widely differing concepts of process, sharing, protection, and communication to run efficiently on the same machine, and to interact in meaningful ways. In addition, the Psyche implementation effort is addressing a host of systems issues for large-scale shared-memory multiprocessors, including remote access to kernel data structures; the organization of kernel address maps; the design of appropriate synchronization and scheduling mechanisms; user-level device drivers, loaders, and pagers; remote source-level kernel debugging; rapid turn-around for the kernel debugging cycle; resource reservation mechanisms for real-time applications; and page migration and replication to maximize locality.

1. Introduction

Parallel processing is in the midst of a transition from special purpose to general purpose use. Part of the impetus for this transition has been the development of practical, large-scale, shared-memory multiprocessors. To make the most effective use of these machines, an operating system must address two fundamental issues that do not arise on uniprocessors. First, the kernel interface must provide the user with greater control over parallel processing abstractions than is customary in a traditional operating system. Second, the kernel must be structured to take advantage of the parallelism and sharing available in the hardware.

If shared-memory multiprocessors are to be used for day-to-day computing, it is important that users be able to program them with whatever style of parallelism is most appropriate for each particular application. To do so they must be able to exercise control over concepts traditionally reserved to the kernel of the operating system, including processes, communication, scheduling, protection, and the grain size of memory sharing. If shared-memory multiprocessors are to be used efficiently, it is also important that the kernel not define abstractions that hide a significant portion of the hardware's functionality.

This work was supported in part by NSF CER grant number DCR-8320136, DARPA ETL contract number DACA76-85-C-0001, ONR Contract number N00014-87-K-0548, and an IBM Faculty Development Award.

The Psyche project is an attempt to design and prototype a high-performance, general-purpose operating system for large-scale shared-memory multiprocessors. The fundamental kernel abstraction, an abstract data object called a *realm*, can be used to implement such diverse mechanisms as monitors, remote procedure calls, buffered message passing, and unconstrained shared memory. Sharing is the default in Psyche; protection is provided only when the user specifically indicates a willingness to sacrifice performance in order to obtain it. Sharing also occurs between the user and the kernel, and helps to enable explicit, user-level control of process structure and scheduling.

Details of the Psyche kernel interface and its rationale have been presented elsewhere [5, 6]. The purpose of the current paper is to outline the kernel structuring issues that we are addressing in our prototype implementation. This is a work-in-progress paper; we have been writing code for about a year, and have recently completed the initial version of the kernel. We are not at a point where we can make definitive statements based on performance measurements or production-quality applications, but we can draw a few conclusions from our implementation experience and from previous projects [2, 3].

Psyche is intended to be portable to a wide range of shared-memory multiprocessors. Our initial implementation is written in C++ and runs on the BBN Butterfly Plus multiprocessor (the hardware base of the GP1000 product line). We have completed the major portions of the kernel and are experimenting with user-level software. In concert with members of the computer vision and planning groups within the department, we have undertaken a major integrated effort in the area of real-time active vision and robotics. The first "toy" program ran in user mode on Psyche in December of 1988. Our first robotics application is expected to be running soon.

Our robotics laboratory includes a custom binocular "head" on the end of a PUMA robot "neck." Images from the robot's "eyes" feed into a special-purpose pipelined image processor. Higher-level vision, planning, and robot control have been implemented on a uniprocessor Sun. Real-time response, however, will require extensive parallelization of these functions. The Butterfly implementation of Psyche provides the platform for this work. Effective implementation of the full range of robot functions will require several different models of parallelism, for which Psyche is ideally suited. In addition, practical experience in the vision lab will provide feedback on the Psyche design.

2. Synopsis of Psyche Abstractions

The Psyche programming model [6] is based on passive data abstractions called *realms*, which include both code and data. The code constitutes a protocol for manipulating the data and for scheduling threads of control. Invocation of protocol operations is the principal mechanism for accessing shared memory, thereby implementing interprocess communication.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call, termed *optimized* invocation, or as safe as a remote procedure call between heavyweight processes, termed *protected* invocation. Unless the caller insists on protection (by performing an explicit kernel call), both forms of invocation are initiated by an ordinary jump-to-subroutine instruction. In the case of a protected invocation the instruction causes a page fault which allows the kernel to intervene.

To permit sharing of arbitrary realms at run time, Psyche arranges for all realms to reside in a uniform address space. The use of uniform addressing allows processes to share data structures and pointers without the need to translate between address spaces. Realms that are known to contain only private data can overlap, as can realms that are only accessed using protected invocations, so normal operating system workloads will fit within the Psyche address space.

At any moment in time, only a small portion of the Psyche uniform address space is accessible to a given process. Every Psyche process executes within a *protection domain*, an execution

environment that denotes the set of available rights. A protection domain's view of the Psyche address space, embodied by a hardware page table, contains those realms for which the right to perform optimized invocations has been demonstrated to the kernel. A process moves between protection domains, inheriting a new view of the address space and the corresponding set of rights, by performing protected invocations.

In order to execute processes inside a given protection domain, the user must ask the kernel to create a collection of *virtual processors* to be associated with that domain. The kernel keeps track of which processes have moved from one protection domain to another, but aside from this it deals only with virtual processors, leaving the job of process management to user-level code. Users, for their part, need not worry about virtual processors. Above the level of the kernel interface Psyche behaves as if there were one physical processor for each virtual processor. We refer to a virtual processor as an *activation* of a protection domain.

On each node of the physical machine, the kernel time-slices between activations currently located on its node. A data structure shared between the kernel and the user contains an indication of which process is being served by the current activation. This indication can be changed in user code, so it is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space. The kernel's help is not required to create or destroy processes within a single protection domain, or to perform context switches between those processes.

Communication from the kernel to the activations takes the form of signals, or *upcalls*, that resemble software interrupts. Upcalls occur when a process moves to a new protection domain, when it returns, and whenever an error occurs. In addition, user-level code can establish upcall handlers for wall time and interval timers, and can arrange to receive a warning in advance of activation preemption.

3. Kernel Organization

3.1. Basic Kernel Structure

The Psyche kernel interface is designed to take maximum advantage of shared-memory architectures. Since we are interested in concepts that scale, we assume that Psyche will be implemented on NUMA (non-uniform memory access) machines. A NUMA host is modeled as a collection of *clusters*, each of which comprises processors and memories with identical locality characteristics. A Sequent or Encore machine consists of a single cluster. On a Butterfly, each node is a cluster unto itself. The proposed Encore Ultramax [7] would consist of non-trivial clusters.

Our most basic kernel design decisions have been adopted with an eye toward efficient use of very large NUMA machines.

(1) The kernel is *symmetric*. Each cluster contains a separate copy of the bulk of the kernel code, and each processor executes this code independently. Scheduling and memory-management data structures are allocated in the kernel on a per-cluster basis. Kernel functions are performed locally whenever possible. The only exceptions are interrupt handlers (which must be located where the interrupts occur) and some virtual memory daemons which consume fewer resources when run on a global basis.

(2) The kernel makes extensive use of shared memory to communicate between processors, both within and between clusters. Ready lists, for example, are manipulated remotely in order to implement protected invocations. The alternative, a message-passing scheme in which instances of the kernel would be asked to perform the manipulations themselves, was rejected as overly expensive. Most modifications to remote data structures can be performed asynchronously; the remote kernel will notice them the next time the data is read.

Synchronous inter-kernel interrupts are used for I/O, remote TLB invalidation, and insertion of high-priority processes in ready queues.

(3) The kernel operates in two separate but overlapping address spaces. Since each instance of the kernel must be able to interact with each other instance, scalability dictates that a large amount of address space be devoted to kernel data structures. Since the kernel also shares data structures with the user, the entire Psyche uniform address space must be visible to the kernel as well. No available machine provides enough virtual address space for both of these needs. We have therefore designed a two-address-space kernel organization (see figure 1). The code and data of the local kernel instance are mapped into the same locations in both address spaces, making switches between those spaces easy. The user/kernel address space also contains all of user space, and the kernel/kernel address space contains the data of every kernel instance. Local data appear at two different locations in the kernel/kernel space.

As in most modern O. S. implementations, little distinction is made between parallelism in user space and parallelism in the kernel. Kernel resources are represented by parallel-access data structures, not by active processes. An activation that traps into the kernel enters a privileged hardware state ("supervisor mode") and begins to execute trusted code, but continues to be the same active entity that it was in user space.

When executing in user space, the activations of separate protection domains must have separate page tables. The kernel is included in each of these page tables (accessible only in supervisor mode), so that there is in fact a separate user/kernel address space for each protection domain. A disadvantage of this scheme is that address space switches are required not only to access data in the kernel/kernel address space, but also in order for the kernel to examine user data in more than one protection domain (as for example, when invoking a protected realm operation). An alternative would be to provide a single, universal user/kernel address space used on

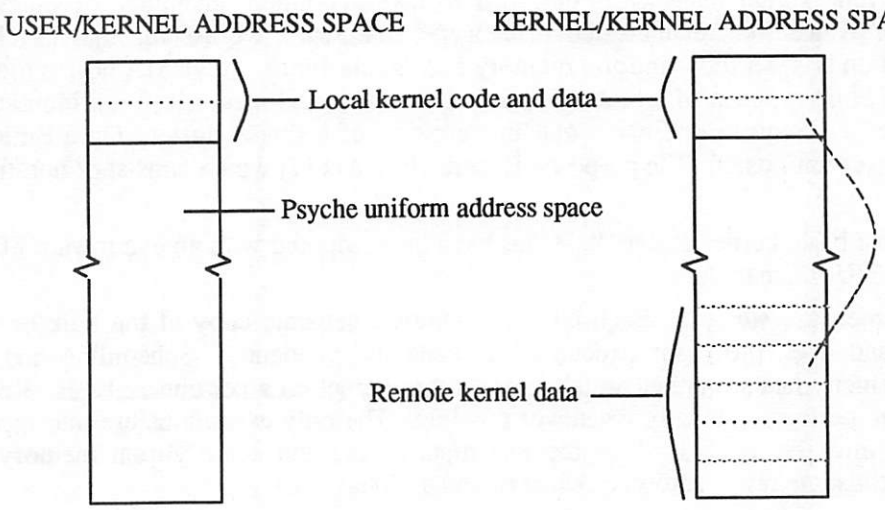


Figure 1: Kernel address spaces

kernel entry by every activation. It is not yet clear whether the resulting savings in address space switches would justify the additional cost of maintaining consistency with user-level page tables.

3.2. Synchronization

A traditional uniprocessor operating system often obtains mutual exclusion for kernel data structures by disabling preemption in the kernel. In a shared-memory multiprocessor, this simple technique no longer works. Data structures can be modified remotely. In fact, an early inventory of kernel data structures in Psyche revealed that almost none (other than those local to a subroutine) was private to a single processor. As a result, explicit synchronization is almost always required when accessing kernel data. We have therefore opted *not* to prohibit preemption in the kernel; there seems to be no point in doing so. The overhead incurred by explicit locking remains to be measured. We expect it to be significant, but our intuition is that it will still be less than the cost of message-passing between kernels to avoid the need for locking.

We have found a need in the kernel for four major types of synchronization. (We also have a facility for all-processor barrier synchronization, but this is used only for kernel initialization.)

disabled preemption
: Those few data structures that *are* processor-local (buffers for the per-processor console, for example) can be protected by disabling preemption. To allow nesting of locks, the kernel maintains a "preemption level" that is incremented when entering a critical section and decremented when leaving. At the end of a quantum, the clock handler forces a context switch only if the counter is zero. If the counter is positive, the handler sets a flag. The code that decrements the preemption level counter causes a context switch on behalf of the clock handler if the flag is set and the level has returned to zero.

locked-out interrupts
: Interrupt masking is used solely to synchronize with device handlers. Data structures shared with devices are never accessed remotely.

spin locks
: Spin locks are the most frequently-used locks in the kernel. There are separate EREW and CREW locks,[1] though the former are much more common. Spin locks are used only to protect critical sections of small, bounded length. The spin lock implementation disables preemption to ensure that the bound is not violated by an inopportune context switch.

scheduler locks
: For those situations in which an activation must wait for a condition that may not happen soon, we provide a simple mechanism to interact with the time-slicing scheduler. Every activation contains in its context block a flag that indicates whether its state has been "saved successfully." To block itself, an activation (1) disables preemption, (2) writes its name down where some other activation will find and resume it at an appropriate time, and (3) invokes the activation scheduler. The scheduler sets the flag of the old activation, clears the flag of the new activation, and re-enables preemption. Anyone who wants to resume an activation must spin until the state-saved flag is set. This mechanism suffices to implement semaphores or monitors and is also used by the clock handler to insert the current activation on the ready list and force preemption. The scheduler always assumes that preemption is disabled and that its caller has done something appropriate with the formerly-running activation.

[1] Exclusive read, exclusive write; concurrent read, exclusive write. EREW locks have lower overhead; they can be acquired and released more quickly than CREW locks in the absence of contention. CREW locks are appropriate when contention is high and reading is much more common than writing.

4. Resource Management

4.1. Devices

On the Butterfly Plus, Psyche supports three classes of devices:

(1) A pair of serial lines connects the "king node" of the Butterfly to a Unix host machine. One of these lines is used by Psyche as a console; the other is used for debugging (see section 5). Both of these uses are encapsulated in the kernel; serial line I/O is not meant to be exported to user-level programs.

(2) Our hardware also allows a Multibus cage to be connected to an individual node. The only Multibus device we support is an Ethernet interface. This is currently used to provide a simple remote file system (for development purposes) and Unix-style standard I/O.

(3) Non-network I/O travels over a VME bus. BBN hardware attaches the bus directly to the Butterfly communication switch, in place of one or two processor nodes. This connection method is superior to that of the Multibus both in terms of potential throughput and in its independence of any particular managing node. In our robotics lab the VME bus is used to communicate with the low-level image processor and the robot eye controllers.

Consistent with the Psyche philosophy of user-level flexibility and kernel minimality, we have developed an interface for memory-mapped I/O devices that limits the kernel's role to basic initialization and forwarding of interrupts. The `make_realm` system call allows the user (with appropriate access rights) to create a realm at a specified virtual or physical address. On the Butterfly, Multibus devices are accessed at special virtual addresses (decoded off the virtual address bus) and VME devices are accessed at the physical addresses corresponding to the VME adapter's location on the communication switch. By creating a memory-mapped realm, a user-level program obtains the ability to read and write device registers without the assistance of the kernel. A second kernel call allows the program to request that device interrupts be translated into upcalls into a user-level activation. We expect these upcalls to be generated with an acceptably small amount of overhead (though obviously more than a simple kernel-level interrupt handler). Again, the actual performance figures have yet to be obtained.

4.2. Virtual Memory

The largest and most sophisticated portion of the kernel is devoted to memory management [1], comprising four distinct abstraction layers. The lowest (NUMA) layer provides an encapsulation of physical page frames and tables. The second (UMA) layer provides the illusion of uniform memory access times through page replication and migration. The third (VUMA) layer provides a default pager for backing store and a mechanism for user-level pagers. The final (PUMA) layer implements Psyche protection domains and upcalls. Page faults may indicate events of interest to any of the layers; they percolate upward until handled.

The PUMA layer maintains a mapping that allows it to identify the realm that contains a given virtual address. This mapping is consulted when a page fault propagates to the PUMA layer, and allows the kernel to determine whether an attempt to touch an inaccessible realm constitutes an error, a protected invocation, or an initial use of something that should be mapped in for optimized access. The UMA layer is strictly divided between policy and mechanism. It is not yet clear how best to decide when to replicate and migrate pages, and this division facilitates experiments. There is no notion of location attached to a realm; the placement of its pages is under the complete control of UMA-layer policies. High-quality policies are likely to depend on the judicious use of hints from user-level software.

The Psyche external pager mechanism is similar in spirit to that provided in Mach [8], but with an interface based on shared memory instead of message passing. The `make_realm` kernel call allows the user to specify a pager activation capable of providing missing pages and

disposing of pages replaced by the kernel. Rather than send the pages in messages, as in Mach, Psyche simply provides the pager with optimized access to the data and code of the realm to be paged. Page-out and page-in requests are provided to the pager as upcalls. Page-out and page-in completion are indicated by the pager with kernel calls. Attempts by the pager to write into non-resident pages of the realm result in page faults that the kernel interprets as anticipatory page-in (pre-paging).

To bootstrap Psyche, the kernel creates a single primordial realm in a single protection domain, containing a single user-level process. This process executes code to create additional realms. Program loaders are outside the kernel, and may be integrated with external pagers. To run a user program, a shell (1) reads the header of the executable file to determine program size, (2) executes kernel calls to create an empty realm and one or more activations, (3) invokes a linker to relocate the executable into the virtual address of the newly-created realm, (4) copies the code and data into the realm, and (5) performs a protected invocation to start an activation running. More realistically, steps (3) and (4) can be replaced by communication with the default or user-provided pager to associate the realm with its executable file and relocation information. Pages can then be supplied on demand, and need not be written (with great amounts of unnecessary paging traffic) at start-up time.

4.3. Support for Real-Time Applications

Though the principal goal of Psyche is to support general-purpose parallel computing, we interpret this goal to include applications for which real-time support is important. We are interested in real-time computing as a research area, and are in the early stages of design work on a real-time subset of Psyche. We are well aware of the difficulties of adding real-time support to a pre-existing operating system, but have several mitigating factors on our side in our attempt to do this in Psyche. First, we are free to change the kernel or its interface when necessary. Second, we are able on a multiprocessor to segregate real-time and non-real-time portions of our workload onto different processors, where they can be managed with different policies. Third, we have in Psyche a kernel that is unusually small and easily adaptable to the needs of user programs.

The segregation of real-time and non-real time processes is facilitated by the already-existing mechanism for creating realms at specified physical locations. We can use this mechanism to dedicate the physical memory of a processor or cluster (without paging) to a particular application. An additional kernel call allows us to dedicate the computational resources of those processors to the activations of the application.

5. Experience with Tools

The Psyche kernel is written in C++ and compiled with the GNU (Free Software Foundation) g++ compiler. We have found the disciplined use of C++ abstractions to be useful in organizing our code. In addition, though this is difficult to quantify, we believe that it has helped to reduce the number of bugs in the code significantly. Unfortunately, the GNU compiler is still undergoing development, and has evolved considerably over the past year. It is unclear whether the advantages of C++ (versus C) have saved us as much time as we have lost to compiler bugs. As stable, high-quality C++ compilers become available, we expect to see them used more and more for operating system development.

One particularly useful language extension provided by the GNU compiler is the ability to redefine the built-in `new` operator and provide it with additional arguments (other than simply the size of the object desired). We have used this extension to provide multiple classes of dynamically-allocated memory. Separate allocators are used for (1) a processor-local heap, (2) the globally-accessible heap, (3) physically-contiguous, non-paged memory for page tables, and (4) physical "page zero" memory required by Butterfly microcoded atomic operations. For cases (2) and (3), an additional argument allows the programmer to specify a preferred node on

which to allocate the memory.

We have also defined a memory pseudo-class called "static" that can be used in conjunction with `new` to specify the time at which the constructor is called for a statically-allocated object. Since the kernel boots in an uninitialized environment (without page tables, interrupt vectors, etc.), we cannot in general allow these constructors to be called immediately upon startup. An additional user-provided argument specifies the virtual address of the object to be "allocated."

When power-cycled, the Butterfly Plus executes a serial-line loader in ROM. For many months we used this ROM directly to load Psyche at 9600 baud. As the kernel grew, this became increasingly painful. To speed the development cycle, we devised a small bootstrap program that initializes the Ethernet interface and then loads the bulk of the kernel using a naive (busy-wait) implementation of UDP. This bootstrap loader was surprisingly easy to write; we wish we had built it sooner.

To facilitate re-execution (for cyclic debugging, for example) we implemented a mechanism to restore the initial state of the kernel on demand. Immediately upon startup, we save a copy of the initialized data segment, and compute a checksum of the code. Upon receipt of a special character sequence, the console line interrupt handler restores the data, verifies that the code is uncorrupted, resets the hardware, and branches to the beginning of the kernel. The cost of this mechanism is small enough, both in complexity and space overhead, to recommend as a general practice for other kernel developers.

The most important tool we have constructed for Psyche is a mechanism for remote, source-level debugging, in the style of the Topaz TeleDebug facility developed at DEC SRC [4]. An interactive front end runs on a Sun workstation using the GNU gdb debugger. Gdb comes with a remote debugging facility; relatively minor modifications were required to get it to work with Psyche. The debugger communicates via UDP with a multiplexor running on the Butterfly's host machine. The multiplexor in turn communicates with a low-level debugging stub (lld) that underlies the Psyche kernel

The multiplexor allows many different debugging sessions to be underway simultaneously, each of them talking to a different Psyche node. It communicates with lld via one of the serial lines connected to the Butterfly king node. The interrupt handler for the debugging line accumulates input until it recognizes a special debugger packet termination character. It looks inside the packet to determine the node for which the packet is intended, and either wakes up the instance of lld on its own node or causes a remote interrupt to effect the same result on another node.

The protocol between gdb and lld is strictly request-reply, and does not require reliable communication. Lld is stateless, or as close to stateless as possible. A debugger can be attached to any instance of the kernel at any time. Lld is also very simple, by design. It was the first portion of the kernel to be written, and has proven extremely useful. With it we are able, for example, to single-step through interrupt drivers using all the facilities of a high-quality source-level debugger.

One question that arises in the design of a remote debugging facility is where to keep track of the instructions that underlie breakpoints. If breakpoint information is kept on the host machine the target system becomes unusable if the debugger crashes. Topaz therefore maintains its breakpoint information in the debugging stub on the target. The guiding philosophy behind this decision is that it should always be possible to debug, so long as the debugging stub remains intact. For the sake of simplicity, we initially kept our breakpoints on the Sun. Lld tended to break more often than gdb anyway, and only infrequently did we find ourselves unable to continue debugging because of lost information. As the kernel has become more stable and our debugging needs more sophisticated, this situation has begun to change. Particularly annoying is the fact that the kernel cannot be restarted if its code has been corrupted by breakpoint trap

instructions. We are now in the process of moving breakpoint data into lld. Only the underlying instructions will be maintained; associated conditions, commands, enable status, etc. will still be kept in gdb.

We are also currently working on mechanisms to extend the benefits of remote debugging up into user-level programs. Of particular interest as a research issue is the appropriate *focus* of debugging. In Psyche, a human user may wish to debug a process, a realm, or (more nebulously) an entire application. We expect special facilities to be needed to address these different views. When debugging a process, for example, breakpoint traps should be ignored when encountered by a process not currently at the focus of attention. Since breakpoints manifest themselves as kernel traps, the kernel will need to share more semantic information with user-level debuggers than is customary in traditional debugging systems.

6. Conclusion

An evaluation of the Psyche user interface from the programmer's point of view will depend on experience with applications. An evaluation of its implications for kernel performance will require more tuning and measurement than we have been able to undertake to date. We intend to focus in particular on the cost of protected procedure calls, page fault handing (which subsumes communication as well as virtual memory in Psyche), and the generation of upcalls for events such as I/O and timer expiration.

In the implementation, we have been happy with the modularity and structure afforded by the symmetric, shared memory organization of the kernel and the use of C++. We have also found that the layering of the VM system makes it relatively easy to understand and modify. Remote debugging at the lowest levels of the kernel has been extremely valuable, as have the mechanisms for Ethernet loading and software kernel restart.

Some of the costs of our implementation decisions have yet to be fully evaluated. One potential source of overhead is the frequent use of locks for synchronization of access to data structures shared between nodes. Another is the memory management context switches induced by the two-address-space structure of the kernel. A third is the propagation of page faults through an explicitly layered VM system. Each of these will be the focus of study as the kernel matures.

Partly as a result of our experience with previous versions of the BBN Butterfly [3] and partly as a result of our work to date on Psyche, we are able to say a number of things about the design of the Butterfly Plus. We are pleased with the machine in most regards. It is the only commercially-available shared-memory MIMD multiprocessor that will scale to large numbers of nodes. In our estimation, this makes it the most attractive machine on the market for research in parallel operating systems.

The Butterfly displays no noticeable switch contention, though memory hot spots of course present a problem. The I/O potential of the VME adapter is very good — much better than that of the processor-local Multibus adapter. It would be useful to be able to perform DMA directly from the VME bus into the memory of individual processors. As currently designed, data must be copied out of the VME adapter explicitly.

The Motorola 68851 memory management unit is extremely flexible, but suffers from a few annoying problems. Its hierarchical page tables provide very good support for Psyche-style sparse address spaces, but its use of physical addresses for page table pointers makes it inconvenient to walk the tree manually, and makes paging of page tables essentially impossible. Another serious problem for Psyche is that memory cannot be made readable in user mode and both readable and writable in supervisor mode without duplicating page table entries. Finally, a deficiency in the handling of TLB misses during read-modify-write cycles can lead to bus errors when performing 68020 atomic operations. It is cumbersome to handle these in software.

The Butterfly does not support remote invocation of 68020 atomic operations. It provides its own collection of remote atomic fetch-and-phi operations in microcode, and these have proven very useful, though incomplete. A few of the more useful primitives (32-bit fetch-and-store, for example) are missing. Special functions such as the atomic operations and Multibus I/O are invoked by reading and writing special *virtual* locations. This mechanism allows the entire physical address space to be reserved for genuine memory, but introduces a level of memory management complexity that we would have been glad to avoid.

Recent developments in Psyche include the implementation of a simple command shell, remote file access via Ethernet, the VME driver, and a linker/loader for user programs. We expect soon to demonstrate a fully-functional kernel by executing our first robotics application, a balloon juggling program that uses VME and Ethernet communication to control our robot eyes and arm. At that point, we plan to suspend the development of new facilities for a short time while we reorganize and evaluate our current implementation.

References

[1] LeBlanc, T. J., B. D. Marsh, and M. L. Scott, "Memory Management for Large-Scale NUMA Multiprocessors," Technical Report, Computer Science Department, University of Rochester, 1989.

[2] LeBlanc, T. J., J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl, and P. C. Dibble, "Experiences in the Development of a Multiprocessor Operating System," *Software — Practice and Experience*, to appear, 1989.

[3] LeBlanc, T. J., M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages, and Systems*, 19-21 July 1988, pp. 161-172.

[4] Redell, D., "Experience with Topaz TeleDebugging," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pp. 35-44. In *ACM SIGPLAN Notices 24*:1 (January 1989).

[5] Scott, M. L., T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, 15-19 August 1988, pp. 255-262, vol. II – Software.

[6] Scott, M. L., T. J. LeBlanc, and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," Technical Report, Computer Science Department, University of Rochester, 1989.

[7] Wilson, A. W. Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Fourteenth Annual International Symposium on Computer Architecture*, 2-5 June 1987, pp. 244-252.

[8] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 63-76. In *ACM Operating Systems Review 21*:5.

Experience with PMothra: A Tool for Mutation Based Testing on A Hypercube†

ByoungJu Choi
Aditya P. Mathur

Software Engineering Research Center
Department of Computer Science
Purdue University
W. Lafayette, IN 47907

July 22, 1989

Abstract

Reliable software testing is a time consuming operation. In addition to the time spent by the tester in identifying, locating, and correcting bugs, a significant time is spent in the execution of the program under test and its *instrumented* or *fault induced* variants. When using mutation based testing to achieve high reliability, the number of such variants can be very large. In this paper we describe experience with a software testing tool named **PMothra** that is designed to provide an architecture-transparent interface to a tester. In its current version, **PMothra** exploits the hypercube architecture by scheduling the execution of mutants on a 128-node Ncube/7 hypercube. Benchmarks illustrating the performance characteristics of **PMothra** are presented. Problems faced with the design of such a system are described.

Index terms–Software testing, software reliability, hypercube, MIMD architecture, mutation analysis, Mothra.

1 Introduction

Mutation analysis [6] is a well known technique for software testing. In the past, several attempts have been made to improve the performance of mutation based testing tools by using a vector [13] or a parallel machine [12]. These efforts have essentially been investigations into *how* a mutation based tool would perform if implemented on a given type of machine.

We have designed a tool named **PMothra** that provides a tester the ability to use parallel architectures in a transparent manner. In this paper we describe our experience with the performance of **PMothra**. As **PMothra** is based on an earlier tool named *Mothra* described in [2,5], all the features of *Mothra* are available to the tester. Thus, **PMothra** provides an easy to use, integrated environment for software testing.

The remainder of this paper is organized as follows. In section 3 we argue that parallel machines are useful in software testing. Section 4 provides an overview of **PMothra** architecture. A description of the experiments performed and the benchmarks obtained appears in section 5. Some enhancements currently underway for **PMothra** are mentioned in section 6. Our experience in using the Ncube/7 hypercube, whose architecture is summarized in the next section, led us to formulate certain hardware and software improvements in the machine. These are listed in section 7. We conclude and provide indications of future work in section 8.

2 An Overview of Mutation Analysis

Mutation analysis, hereafter referred to as *mutation*, is a technique for determining the adequacy of a test set. It is a formal procedure that helps a tester decide when to stop testing. In this section we provide an overview of the technique. Details may be found in [2,5,6].

Given a program P under test and a set of test cases $\mathcal{T}$, the question we ask is: *Is $\mathcal{T}$ an adequate test set for P?* Mutation relies on the *competent programmer* hypothesis. According to this hypotheses while programmers develop programs containing faults, these programs are *close* to their correct versions. Research in mutation has led to the design of a set of language specific *mutant operators*. Each mutant operator models a specific fault. For example, the *scalar variable replacement* operator models "use of the wrong variable" fault. A complete set of these operators may be found in [1,15].

When applied on a program, a mutant operator *induces* a fault in P by making a simple syntactic change. The program generated by mutating P is known as a *mutant* of P. For example, if $a := b+c$ is an assignment in P, and a, b, c, and d are the only variables used in P, four of the nine mutants of P created by the scalar variable replacement operator will have the above assignment replaced by: $b := b+c$, $a := a+c$, $a := b+a$, and $a := d+c$. Note that a mutant is identical to P except in the mutated statement.

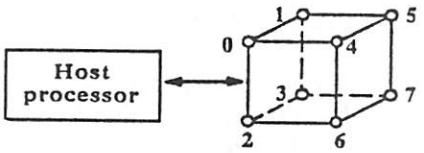
Mutation based testing proceeds in a sequence of phases[1]. In phase I, P is executed on each element of $\mathcal{T}$. We assume that P generates the correct output, also known as the *expected output*, on each element of $\mathcal{T}$. The correctness of the output is decided by an *oracle* which can be the human tester, a program, or a combination thereof. In phase II, each mutant operator is applied on P to generate a set of mutants. Each mutant is executed on elements of $\mathcal{T}$ until either (a) the output of the mutant on a test case $d \in \mathcal{T}$ is different from the output of P on d, or (b) there are no more test cases in $\mathcal{T}$. If (a) is true, the mutant is considered *killed*. If (b) is true the mutant is considered *live*. A mutant might remain live either because it is semantically equivalent to P, or the test data is not adequate, or P has an error.

At the conclusion of mutant executions in phase II, the *mutation score* is computed as the ratio of the total number of mutants killed to the total number of non-equivalent mutants generated. A higher score, close to 1, indicates that $\mathcal{T}$ is close to being adequate. Phase III consists of an examination of live mutants. In this process the tester might detect a program error or generate additional test cases to kill one or more mutants. In case an error is discovered, P needs to be modified and testing resumes from phase I. In case additional test data is generated, phase III continues until either a satisfactory mutation score is obtained or an error is found.

3 Why Use Parallel Machines ?

Semantic faults [14] are defects that arise from programmer errors committed while communicating to a machine the *meaning* of what is to be done. Detection and removal of semantic faults is the prime goal of software testing. As evidenced by the data compiled by Musa et al [14], computer turnaround time is one of the significant factors affecting the effort required to detect and correct faults. The more time required to execute P and its mutants, the more non-productive wait time will be involved in detecting and removing a fault. Studies such as the one involving the DATACOM project [14], clearly indicate that an increase in the available computer time has a significant effect on the reliability of the product. Other studies, such as the one by Shooman et al [16], strengthen this conclusion. It is this conclusion that has been the prime motivation for our research. Providing a software testing tool that can efficiently exploit the architecture of a parallel machine implies providing more computing power to the software tester and hence an opportunity to improve the reliability of the product being developed.

[1] Note that a mutation based tool, such as Mothra [5], might not enforce the sequencing described here.



A message passing 3-cube architecture.
A circle denotes a node, containing a proc
essor-memory pair.

Figure 1: A message passing local memory hypercube.

The cost of hardware has been declining rapidly. Low cost parallel machines, that can be configured as a hypercube or in several other topologies, are available as add-ons to workstations [17]. We believe that the work reported here will provide the motivate software engineers to use powerful hardware for improving the reliability of their software through exhaustive tests. By exhaustive tests we mean tests conducted so as to obtain a mutation score of near 99 %. Some of the microprocessors announced recently, notably the 64-bit i860[2] chip from Intel [11], have the potential to outperform the traditional supercomputers when used in a tightly coupled multiprocessor configuration. For example, on the Linpack benchmark, a single processor i860 based system performs at approximately one third that of a Cray X-MP[3] single processor machine [11]. Thus, the i860 has the potential of providing extremely high performance, at an affordable price, on a workstation. A tool such as **PMothra** enables a software tester to exploit such power without any reprogramming effort.

The machine architecture currently supported by **PMothra** is shown in Fig. 1. We assume the availability of a pool of processor-memory (*P-M*) pairs. Each *P-M* pair can communicate with the others in the pool using a communication network. In a hypercube, such a network is an n-dimensional cube having $N = 2^n$ *nodes*. Each *P-M* constitutes a distinct node and has exactly n *near* neighbors with *direct* communication links. Thus, routing a message from a node to its near neighbor requires the traversal of exactly one communication link. Routing a message from one node to any other node in the hypercube requires the traversal of at most n links. Fig. 1 shows a 3-dimensional hypercube consisting of 8 *P-M* pairs. We often refer to all such pairs as a *cube*.

4 Structure of PMothra: An Overview

In this section we provide a brief overview of the structure of **PMothra**. Details of the architecture may be found in [3].

PMothra is based on *Mothra* [5]. In addition to all the facilities provided by *Mothra*, **PMothra** provides the tester the ability to select one out of many computing resources. A tester has the option of selecting a hypercube computer, namely the Ncube/7 [10], a vector-multiprocessor, namely the Alliant FX/8 [7], and the SUN workstation (this is the default option). Once the resource is selected, **PMothra** provides the tester with a transparent interface to the underlying machine. Without any additional effort of having to learn the programming and architectural details of the underlying machine, the tester gets its speed advantage. Thus, **PMothra** can be treated as a computational resource manager of *Mothra*.

[2] i860 is a trademark of Intel Corporation.
[3] Cray X-MP is a trademark of Cray Research Inc.

In the current version, the scheduler, a key component of PMothra, supports the hypercube architecture. Its goal is to schedule the execution of mutants[4] on the nodes of the hypercube. While doing so, it attempts to maximize the speedup and minimize the time to complete the execution of all the mutants.

The scheduler is a process that runs in a specific hardware and software environment. The hardware environment consists of the host, shown in Fig. 1, which is responsible for communicating with the processors on the nodes of the hypercube. In the present implementation, the host is a Sun 3/60 workstation. While running on the host, the scheduler schedules mutants for execution on the next available *P-M* pair from the pool shown in Fig. 1(a).

The software environment of the scheduler is a collection of support processes shown in Fig. 2. There are several processes that belong to the testing tool but are not shown in Fig. 2. For example, the test case editor and mutant status reporter are two of the several *Mothra* related processes not shown here. These processes are described in [2,5].

The scheduler obtains *node modules* from a node module generator. A node module consists of one mutant packaged with an interface for the test case and expected output server. A sample node module, constructed out of a simple Fortran program, appears in the Appendix. Statements included between consecutive pairs of lines with asterisks are the ones added to the original test program for execution on the hypercube. The first set of statements, starting at the declaration of variable *moto* and ending at statement *ie=nread(...)* is the Test case and expected Output Server (TOS), described in section 4.2, interface. The second set of statements, starting from *if (N. ne. ..).*. and ending at *ie=nprint(...)* is the oracle. In the current implementation, these statements are added by a shell script based on the list of input and output variables provided by the tester.

Depending on the nature of the interface, node modules can be of different types as described in [3]. A *node module* is a program that executes on one node of the hypercube. Type L node module, consists of a mutant, a Local Test Case Server (LTCS), an oracle, all test cases $\mathcal{T}$, and all expected outputs $\cup_{d \in \mathcal{T}} P(d)$. Once such a node module is scheduled for execution on a node it continues execution until any one of these conditions is satisfied: (a) the enclosed mutant is killed, (b) test data is exhausted, and (c) the mutant is a runaway process and is therefore killed by the scheduler.

Type G module consists of the mutant M, a Test case and expected Output Server Interface (TOSI), and an oracle. Once such a module is scheduled for execution on a node, it behaves like the node module of type L except that it obtains the test case, and the expected output, from a server. Such a server is located either on the host or on a node different from the one on which the node module is located.

In this paper we consider node modules of type G that are the ones that rely on a *global* TOS for obtaining the test case and the expected output. Thus, when the mutant in a node module is ready for execution, it requests the TOS, through an interface, for the *next* test case and expected output. On receipt of the requested data, the mutant begins execution. On completion of the execution of the mutant, the output generated is compared with the expected output. If these two are different then the mutant is assumed to be *killed* and the node module releases the node on which it was executing. If the outputs are the same, implying that the mutant is *live*, then the node module requests another test case and the corresponding expected output pair. If at least one more such pair is available then the above outlined process is repeated, otherwise the node module releases the node.

At the end of its execution, a node module informs the status update procedure whether the mutant is killed or not. This information is saved in the mutant data base[5]. The data base in turn is used by status reporting tools in *Mothra* to inform the tester about the status of a testing experiment.

[4] A *mutant* is obtained by inducing a simple change in the program P under test. For details see [1,2,6].

[5] This data base is created and maintained by *Mothra*.

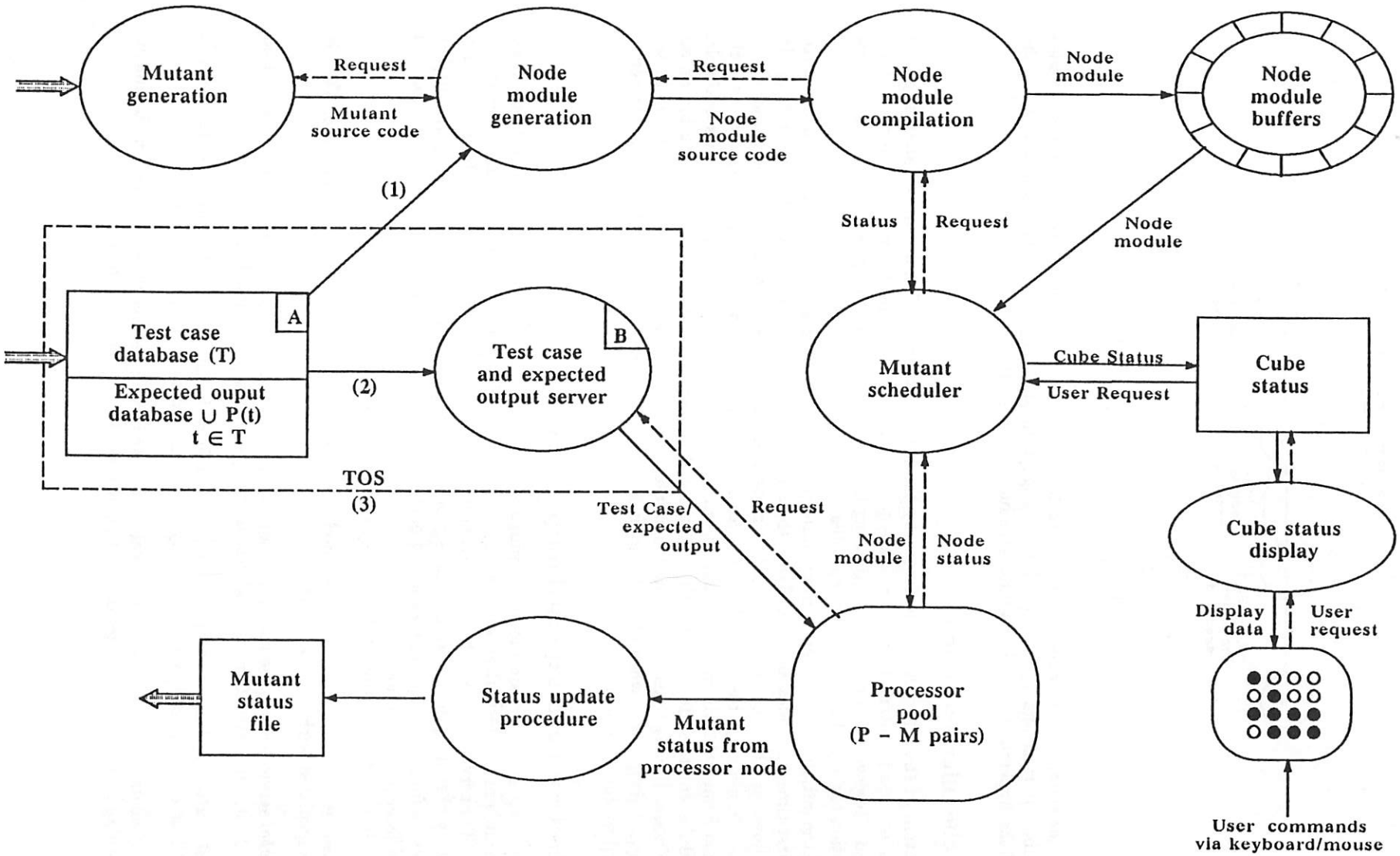


Figure 2: Scheduler and suport processes in P^{M}othra

: To Mothra
: From Mothra
: Process
: Data file
: Busy node
: Free node

(1) Only when local servers are used.
(2) Only when global server(s) are used.
(3) Both A and B reside on one or more processors in the pool.

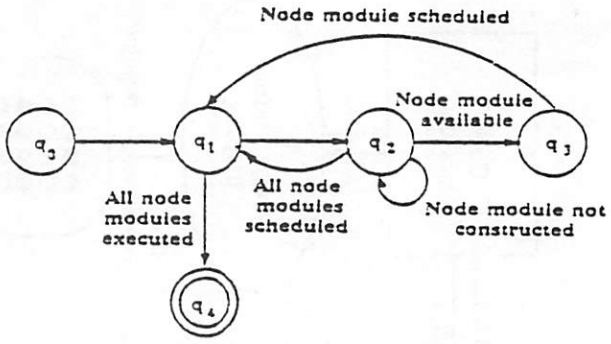


Figure 3: State diagram of the scheduler.

The *cube status* reporting tool shown in Fig. 2 is an independent tool to track the status of nodes in the cube. It runs under the X-windows environment and keeps the tester informed about the status of the hypercube and the mutants scheduled for execution.

4.1 Scheduler execution

The function of the scheduler is illustrated by the state diagram in Fig. 3. Immediately after the scheduler process is initiated, it enters state q_0. In this state, certain initialization operations are performed. Reserving the cube and initializing the status of all nodes to *free* are two of the several initialization tasks performed by the scheduler.

Once the initialization is over, it enters state q_1 where it waits for a node to be free. When entered for the first time, all nodes are free. Hence the scheduler almost immediately moves to state q_2. In general, however, the scheduler may need to remain in state q_1 while waiting for a node module to terminate. In state q_2, the scheduler waits for a node module to be available. The node module is obtained from the node module buffer shown in Fig. 2. Once a node module becomes available, the scheduler enters state q_3 in which it actually schedules the node module for execution on an available node. Having scheduled the node module, it returns to state q_1. The scheduler terminates in state q_4. This happens when all node modules have completed execution. At this time, the cube is released by the scheduler.

4.2 Test case and expected output servers

During its execution, a node module requires a test case d and the expected output $P(d)$. This requirement is met in PMothra with the aid of TOS. A TOS consists of three components as shown in Fig. 4. The server component receives requests a test case or an expected output. It services such a request by obtaining data from the test set or the set of expected outputs depending on whether the request is for a test case or an expected output. All service is requested using a service protocol described later in this section.

Depending on the location of TOS, three service environments are possible:

1. *Local* service, as in the case when node modules of type L are used. In this case, TOS is encapsulated within a node module.
2. *Global* service, when the host processor provides the service. In this case, TOS is located on the host. Only type G node modules can be used when using global service.
3. *Distributed* service, when one or more nodes provide the service. In this case, the TOS is replicated on one or more nodes. Only type G modules can be used.

The experiments reported in this paper were conducted using the distributed service environment. The advantages and disadvantages of other environments are discussed in [3].

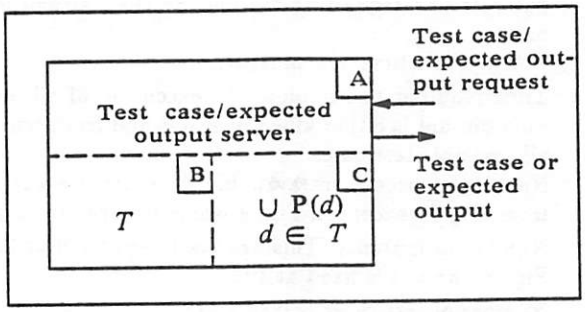


Figure 4: Components of TOS.

5 Experiments with PMothra

In this section we present results of some experiments that are representative of the performance of PMothra on the Ncube/7 hypercube. Three attributes were used to characterize the performance of PMothra: *speedup*, *efficiency*, and *time to completion*. These attributes are defined in Fig. 5. One might envision the speedup to be almost linear in our application because the processes that execute at each node, namely the mutants, are independent of each other and hence do not require any communication over the inter-connection network of the hypercube. As shown below, we do get speedups that are almost linear. However, the following factors are responsible for the deviation from the expected linearity:

1. *Loading overhead:* This is the time required to load each mutant on the node of the hypercube.
2. *TOS service time*: Each time a mutant requests a test case and expected output from a TOS, it encounters a delay that is the sum of the following times:
   (a) Time required to communicate the test case and expected output to the mutant over the inter-connection network. For a given machine, this time depends on the number of bytes transferred and the traffic on the inter-connection network. The size of a test case depends on the program being tested and the environment of its intended use. For example, a sort program could be tested on a 10K byte array or only a 10 byte array. However, the size does affect the optimum number of TOS in a system.
   (b) Time required for the TOS to service requests from mutants, executing on other nodes, that have already arrived. This depends on the number of mutants being served by the TOS and the execution rate of these mutants.

If the time delays mentioned were absent, then we would have an *ideal* environment. In such a situation, our system could provide linear speedup. It is this *ideal*, or maximum possible speedup, that acts as a reference point for evaluating the speedup obtained via experiments.

To understand the *time to completion*, T_N, we note that it is only the *live* mutants that are executed. As an example, consider that there are two mutants and two test cases denoted by d_1 and d_2. If these mutants are executed first on d_1 and get killed, then the total number of mutant executions is two. On the contrary, if none of these two mutants is killed by d_1, then the total number of mutant executions is four irrespective of whether the mutants are killed by d_2 or not. PMothra uses a simple mutant scheduling scheme according to which a mutant is executed on only one test case at a time. Once its execution is over, if it remains alive then it is executed on the

t	Average execution time of a mutant.
s	Average time required to send one test case and expected output pair to a requesting mutant. Set to 1 second in all experiments.
c	Compilation time of a mutant.
T	Time required to complete the execution of all mutants until each mutant is either killed or is live and has been executed on all available test cases.
N	Number of processors available for mutant execution. Varies from 1 to 63 except in Fig. 9 where it varies from 1 to 128.
M	Number of mutants. This has been kept fixed at 500 except in Fig. 9 where it is fixed at 128.
η	Number of test cases. Fixed at 1.
T_N	Time to execute all the mutants against all test cases on N processors.
T_1	Time to execute all the mutants against all test cases on 1 processor.
Speedup	$\frac{T_1}{T_N}$
Efficiency	$\frac{\text{speedup}}{N}$

Figure 5: Symbols and definitions used in the description of experiments.

next available test case. This scheme avoids *redundant* execution of mutants that might result if a mutant is executed on more than one test case concurrently. The time to completion, denoted by T, is defined as the total time to execute all mutants until each mutant is either killed or test data is exhausted.

The experiments were designed to answer the following questions:

1. How does the number of TOS's affect the speedup that can be obtained using **PMothra**?
2. What is the optimum number of TOS's?
3. How does the time to complete the execution of all mutants vary as the number of TOS's is increased?
4. What is the effect of compilation time on the speedup?

We now present the results of different experiments. Fig. 5 summarizes the terminology and values of different parameters used in the experiments. All experiments were performed on the Ncube/7 machine consisting of 64 $P-M$ pairs. The data shown in Fig. 6, 7, and 8 was generated by varying t. This variation was achieved by increasing the value of a loop variable in each mutant so that the mutant execution assumes the desired value of t.

Speedup versus the number of TOS

Fig. 6 shows how the speedup varies as the number of TOS is increased. As the number of TOS is increased, the speedup increases because of a decrease in the mean waiting time of each mutant for receiving the test cases and expected output. However, increasing TOS decreases the number of processors available for mutant execution. Thus, the speedup peaks at the *optimum* number of TOS and then begins to decline. Notice that the peak, and hence the optimum number of TOS, is dependent on t which is the average execution time of a mutant. The peak occurs at smaller values

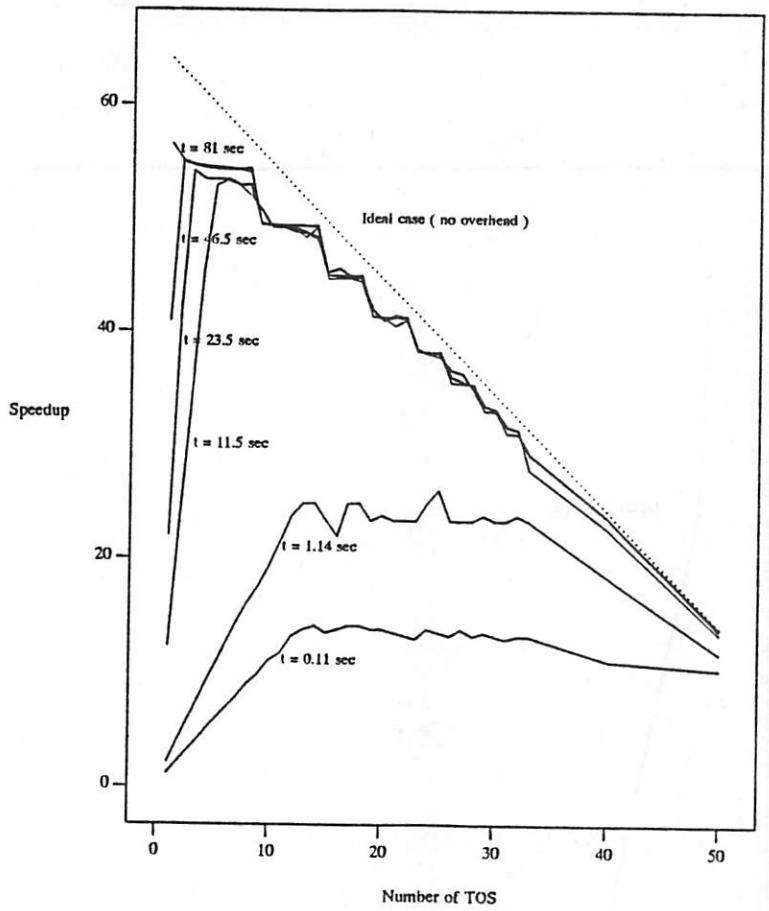


Figure 6: Effect of number of servers on the speedup.

as t increases. The reason for this behavior should be obvious. An increase in t implies a decrease in the number of requests from a mutant during a given time period. Hence, lesser number of TOS are required.

In Fig. 6, the dotted line shows the maximum possible speedup (ideal) under ideal conditions described earlier. Note that the speedup obtained using PMothra approaches the ideal value as t increases. For $t = 0.11$ seconds the maximum speedup obtained is 16 though the speedup obtainable under ideal conditions is 52. However, when t increases to 81 seconds, the maximum speedup obtained is 55 as compared to the maximum attainable 62. Thus, we see that the efficiency with which our algorithm uses the machine increases as t increases. If computed using the data from Fig. 6, the efficiency varies between 0.25 to 0.86. The reason for this behavior can be understood by noting that for a fixed mutant size, an increase in t reduces the relative overhead due to mutant loading and service required for test cases and expected output. Hence, the processors spend most of the time executing mutants rather than waiting to receive a mutant, or a test case, or an expected output.

Optimum number of TOS

Fig. 7 shows the effect of varying the ratio $\frac{t}{s}$. It has been derived from Fig. 6. Note that as t increases relative to s, the number of TOS required to achieve the maximum speedup decreases. Once again this behavior is due to the fact that mutants have to wait less to receive test case and expected output from TOS when t is large relative to s.

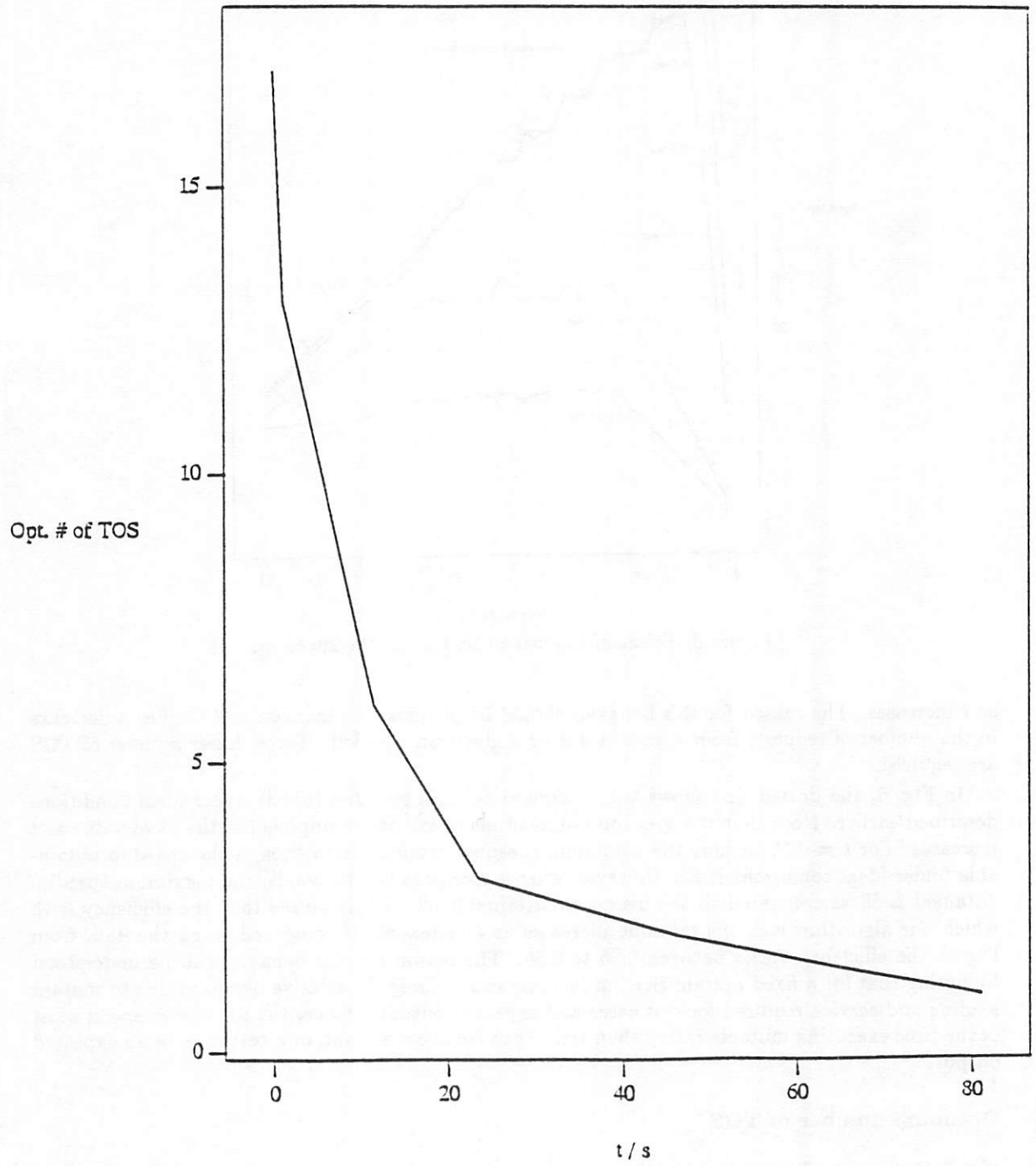


Figure 7: Optimum number of servers.

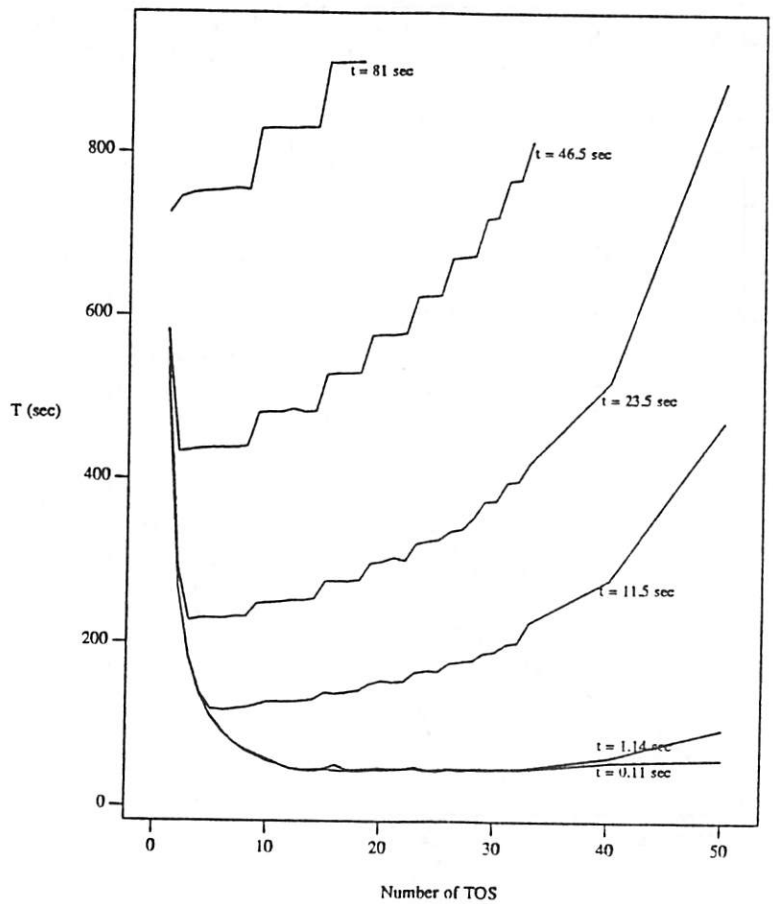


Figure 8: Time to completion vs. number of servers.

Time to completion

Fig. 8 shows how T is affected by the number of TOS. It is easy to predict this behavior from Fig. 6. For each t, there exists an optimum number of TOS that results in the least value of T.

The step-like behavior of the curves in Fig. 8 can be explained when one examines the scheduling strategy closely. The mutants are scheduled in *waves*. The total number of waves is $\lceil \frac{M}{N} \rceil$. Thus, in the first wave, N mutants are scheduled. Assuming that all of them complete approximately together, the next N mutants are scheduled in the second wave, and so on. The number of such waves remains constant over a range of values of N. Note that the change in N is brought about by an increase in the number of TOS. Each TOS occupies one processor, thus reducing the number of processors available for mutant execution.

For example, if we have 500 mutants and $N = 63$, a total of 8 waves are required. For $N = 62$, 9 waves are required. The number of waves required remains 9 until N becomes 55 when it increases to 10. Also note that all mutants scheduled in one wave execute concurrently. Thus, the number of mutants that execute in one wave does not have any significant effect on T, hence the near-horizontal lines in the curves in Fig. 8. However, at each point of change in the number of waves, there is a jump in T brought about by t. Note that the height of each jump is proportional to t.

Compilation time and speedup

As shown earlier in Fig. 2, the compilation of mutants is a serial process. In the current implementation, compilation of node modules is carried out on the host processor. It is easy to perceive that node modules that have large compilation times relative to their execution time can lead to low speedup and, consequently, poor efficiency of machine use. Fig. 9 shows the effect of compi-

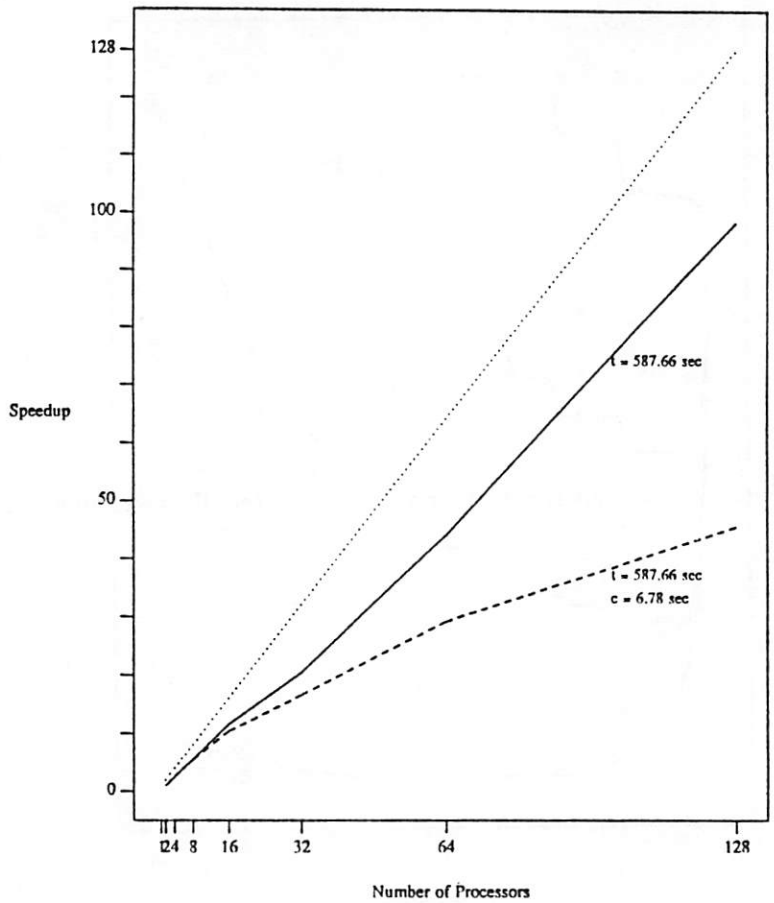


Figure 9: Effect of compilation time on speedup.

lation time on the speedup obtained. As shown, if the compilation time is excluded, the speedup approaches the ideal case, shown by dotted lines. However, for 128 processors, the compilation time of mutants reduces the speedup from 98 to 45, a reduction of approximately 46%.

5.1 Use of experimental results

It is quite clear from the above results that mutation based testing on a parallel machine is efficient when the time to execute mutants is relatively large as compared to the compilation time of node modules. In general, if $c > k \times t$, where k is the average number of test cases required to kill a mutant, the use of a parallel machine becomes extremely inefficient. This is because when a mutant has been compiled and is ready for execution, the previous mutant has completed execution and hence released the processor. Thus, when $c > k \times t$, an average of 1 processor will be busy at any time during testing. However, as c reduces relative to $k \times t$, the machine utilization and the speedup increase.

The graphs presented earlier can be used in practice to decide a) if the parallel machine should be used or not and b) the number of test case and expected output servers that must be employed in case the parallel machine *is* used.

As one of our goals in the design of **PMothra** is to provide a platform for testing large programs, the finding that speedup increases with increase in mutant execution time is significant. It implies that the efficiency at which the hypercube operates indeed increases as the program execution time increases. Notice that we are using the attribute *large* (in terms of source lines of code) to imply large execution time.

5.2 Future experiments

The results reported in this paper are the outcome of the initial phase of benchmarking PMothra after its implementation. A large number of experiments are currently underway. More specifically, these experiments are designed to study the behavior of PMothra for different execution time distributions of mutants of multiple test cases. In addition, we are also interested in studying the precise effect of *test case effectiveness* [15] on the number of TOS and speedup. A theoretical model of the scheduling process is also under construction. We expect such a model to be useful in predicting the number of servers to use and the completion time of mutant execution (T). These estimates will provide the tester information to plan the time and cost of performing a test for a given program.

6 Enhancements to PMothra

PMothra provides an easy to use environment for a software tester. However, in its current version, it fails to meet one of its primary design goals: *transparency of architecture to the tester*. Such a transparency implies that the tester be able to submit a program P under test to PMothra and be able to use the available hardware supported by PMothra without *any* modification in P. In the current version, the tester needs to modify P so that it can execute on the hypercube. Even though this modification, illustrated in the Appendix, may be considered trivial by many a tester, we believe that it should be performed automatically by PMothra and thereby avoid the need to learn the idiosyncrasies of the hypercube.

In the future version, the following tasks are targets for automation:

1. Making changes in each mutant so that all I/O calls are suitably replaced by I/O calls to the host processor. This change may not be necessary if the future version of the Ncube hardware supports I/O on each node.

2. Generation and addition of the TOS interface to each mutant.

3. Automatic generation of TOS based on the test case and expected output requirements of P as specified by the tester. In the current version of PMothra, TOS is coded by the programmer.

7 Suggested Enhancements of the Ncube/7

During the implementation of PMothra we faced several problems due to the limitations of the Ncube/7 system. These limitations are listed below. We hope that future versions of Ncube/7 hardware and software will remove these limitations.

1. The message passing system in the machine is such that often a process needs to poll all nodes for a message. We propose that an *event* based mechanism be available through the hardware in addition to the current message passing mechanism. Using the event based mechanism, a programmer could easily program tasks such as *send a test case to the next waiting program on a node*. In the current system, the only way to program such a task is to keep polling all the nodes for requests for a test case.

2. The operating system does support sharing of the hypercube by multiple users. However, the sharing is limited. For example, each user must have a dedicated subcube. This limits the number of users severely. For example, in the environment in which we work, there are over a dozen active users of the machine. Occasionally, when a few users *grab* the entire cube, the others are left waiting indefinitely. This can have adverse effect on research productivity. We therefore propose a truly multiprogrammed system.

8 Conclusions and Future Work

In the past, several researchers have shown the effectiveness of mutation based testing. However, experiments conducted using mutation have been limited to relatively small programs. This limitation has been primarily due to the excessive computational requirement of mutation based testing. In this paper we have shown how such a requirement can be met using the hypercube. We expect that the availability of a testing tool such as **PMothra** on a powerful machine will provide a platform for researchers in software testing to conduct experiments with significantly larger and realistic software. Where reliability is of prime concern [8], such a tool can be used very effectively. As mentioned earlier, we are currently conducting experiments to determine the performance of **PMothra** in a realistic software testing environment.

There are several improvements that are contemplated in the design of **PMothra**. The ones that are currently being accorded high priority are:

1. A reduction in the compilation time of the mutants, and hence the time to generate each node module.
2. Use of TOS nodes for mutant execution on a lower priority basis.

Reducing the compilation time: In [9], the use of a compiler as an aid in testing was proposed. We propose to take a similar, though more general, approach to reducing the compilation time. As is evident from our description of the **PMothra** architecture, each mutant is compiled independently. We, however, know that the mutants are similar to each other and to P. Thus, the object code of a mutant differs from that of P only in a few instructions. These instructions correspond to the statement in P that has been mutated. Through an approach that we call *compiler integrated testing*, it is possible to compile P just once, and then generate one *code patch* for each mutant. The code patch is then applied to the object code of P to generate the desired mutant. When this approach is used, only one node module is prepared and broadcast to all nodes in the cube.

We hope to gain two advantages from this approach: (a) reduced compilation time and, hence, the improved availability of node modules, and (b) reduced node module loading time on a node. Notice that as the node modules are identical, they can be broadcast initially by the scheduler to all the nodes. On a hypercube, such a broadcast is of the order of $\log n$. Further, the broadcast is needed exactly once during one test cycle[6] for the program under test.

Utilization of the TOS node: In the current system, a node is either used by the TOS or by a node module. Though we expect a node module to be busy unless it is waiting for an input from the TOS, we do expect that a TOS node will be idle when no request is pending. It is therefore obvious that such a node can multiplex between the execution of TOS and a node module. In order to keep the wait time of the remaining node modules to a minimum, TOS can be assigned a higher priority.

On first thought, one might be skeptical about this approach. However, we note that even when an optimum number of TOS is used, the TOS idle time may be significant. This can happen due to the large variance in mutant execution time. It is this idle time that can be well utilized if a node multiplexes between the TOS and a mutant.

We are currently using **PMothra** to conduct experiments designed to study the error exposing capabilities of mutation analysis on large programs. A tool such as **PMothra** is certainly of significant help in testing C programs particularly because of the large number of mutant operators [1]. We hope such studies to shed more light on the cost effectiveness of mutation analysis for improving a program's reliability.

[6] A test cycle begins when the program P under test is submitted to *Mothra* and ends when either all mutants of P have been executed on the test cases, or some error discovered in P.

Acknowledgements

We would like to thank Ronnie Martin for reviewing a preliminary copy of this report, to Rich DeMillo, Ed Krauser, and Vernon Rego for the useful discussions, and to Elizabeth Northern for the figures.

References

[1] H. Agrawal, R.A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur and E.H. Spafford, "Design of Mutant Operators for the C Programming Language," *Technical Report*, SERC-TR-41-P 1989, Software Engineering Research Center, Purdue University.

[2] B.J. Choi, R. A. DeMillo, E.W. Krauser, A.P. Mathur, R.J. Martin, A.J. Offutt, H. Pan, and E.H. Spafford, "The Mothra Toolset," *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, January 3-6, 1989.

[3] B. J. Choi, A. P. Mathur, and B. Pattison, "Architecture of PMothra: A Tool for Mutation Based testing on the Hypercube," Technical Report, SERC-TR-45-P 1989, Software Engineering Research Center, Purdue University.

[4] *CFT77 Reference Manual*, Cray Research Inc., MN 55120, Sept. 1986.

[5] R.A. DeMillo, D.S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An Extended Overview of the Mothra Testing Environment," *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July 1988.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, Vol. 11, no. 4, April 1978.

[7] FX/Series Architecture Manual, Alliant Computer Systems Corporation, Littleton, Massachusetts.

[8] W. Geiger, L. Gmeiner, H. Trauboth, and U. Voges, "Program Testing Techniques for Nuclear Reactor Protection Systems," *Computer*, August 1979, pp10-18.

[9] R. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. on Software Engineering*, VOL. SE-3, NO. 4, July 1977.

[10] J. P. Hayes, T. N. Mudge, and Q. F. Stout, S. Colley, and J. Palmer, "A Microprocessor Based Hypercube Supercomputer," *IEEE Micro*, October 1986, pp6-17.

[11] Intel, "Introducing the Intel i860 64-bit Microprocessor," Intel Corporation, Santa Clara, California, 1989.

[12] E. W. Krauser, A.P. Mathur, and V. Rego, "High Performance Testing On SIMD Machines", *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July 19-21.

[13] A. P. Mathur and E.W. Krauser, "Modeling Mutation on a Vector Processor," *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 11-15, 1988.

[14] J. D. Musa, A. Iannino, and K. Okumoto, "Software Reliability, Measurement, Prediction, Application," McGraw-Hill Book Company, New York, 1987.

[15] A. J. Offutt, "Automatic Test Data Generation," PhD thesis, Georgia Institute of Technology, 1988. GIT-ICS-88/28. (Also released as SERC-TR-25-P, Software Engineering Research Center, Purdue University).

[16] M. L. Shooman and M.I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," *Proc. of 1975 International Conference on Reliable Software*, Los Angeles, pp347-357.

[17] Topologix, "Topology 1000 Product Overview," Topologix Incorporated, Denver, Colorado, 1987.

Appendix

Sample Test Case and Expected Output Server

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C     Sum -    Computes the sum of N integers
C              Input N, Output RESULT
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM SUM

      INTEGER N, RESULT

c|*****************************************************************|
cv                                                                  v
      integer moto(2), Mottc,  Buf(2), MotResult
c             moto(1) is 0 , mutant is alive
c                        1 , mutant is killed
c             moto(2) is test case number which kill mutant
c             Mottc is test case number.
c             Buf(1) is input test case (N).
c             Buf(2) is expected output (RESULT).

      call whoami(idnode, iproc, idhost, idim)
      MotResult = 0
      Buf(2) = 0

9999  continue
      Mottc = Mottc + 1

c     Decide TOS node id among 5 TOS  nodes
      iin = mod(idnode, 5)
      ie = nwrite( Mottc, 4, iin, 2000, ifla)
      ie = nread(Buf, 8, iin, 3000, ifla)
      N = Muf(1)
c^                                                                  ^
c|*****************************************************************|

C     Loop through values and compute sum
      RESULT = 0
      DO 10 I = 1, N, 1
         RESULT = RESULT + I
10    CONTINUE

C     PRINT *, RESULT

c|*****************************************************************|
cv                                                                  v
c     Compare output from Expected output
      if (N .ne. Buf(2))  MotResult = 1

      if ((MotResult .eq. 0) .and. (Buf(2) .lt. MotN)) then
          goto 9999
      endif

      moto(1) = MotResult
      moto(2) = Mottc

c     Send message to host.
      ie = nprint(idhost, 0, '%1D %3D', moto(1), moto(2))
c^                                                                  ^
c|*****************************************************************|
      STOP
      END
```

Debugging and Performance Monitoring in HPC/VORX

Howard P. Katseff

AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

HPC/VORX is a computing system that provides closely coupled computing between large numbers of processors. It also supports the connection of many host workstations which may be geographically distributed within the area of a large building and allows a single application to span many processors and many workstations. The debugging tools available in HPC/VORX range from a traditional symbolic debugger that can be attached to any process of an application to a communications debugger that provides a global view of the state of the multiprocess application. To help a programmer visualize the performance of a distributed application, HPC/VORX includes a tool that provides a visual representation of the execution characteristics of the application in real-time.

1. Introduction

HPC/VORX is a *local area multicomputer* system that combines the major strengths of multiprocessor computer systems and local area networks [1]. Like multicomputers, it exhibits low latency communications, allowing the close cooperation of many processors on a single large application. It also provides for the connection of workstations and other resources that are distributed within the area of a large building, but with better communications performance than is usually found in local-area networks. The current system connects ten SUN 3 workstations and 70 adjunct processors based on the Motorola 68020 and has been operational since early 1988. The system can easily be expanded to more than a thousand nodes by replicating the interconnect hardware.

The HPC/VORX system is based on a high bandwidth, low latency interconnect called the HPC and is controlled by the VORX distributed operating system. A conceptual diagram of HPC/VORX is shown in Figure 1. The right side of the diagram shows resources normally found in a local area network and the left side is the adjunct processor pool that is used for compute intensive or closely coupled medium grain parallel applications. Applications on the adjunct processors may be controlled from any workstation and it is possible to build a single application that spans many workstations and many adjunct processors. HPC/VORX has proven to be a useful base for implementing a variety of applications. Applications implemented on HPC/VORX range from the Rapport multimedia conferencing system [2] to several circuit

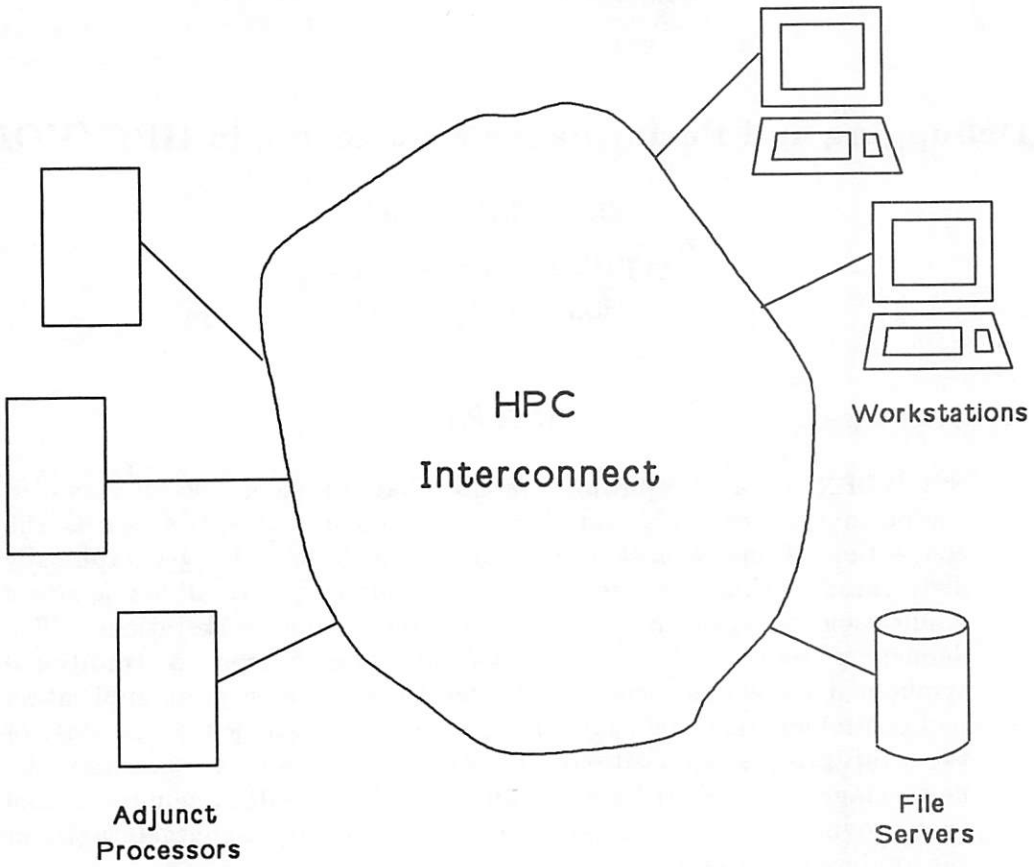


Figure 1. A Typical Local Area Multiprocessor System

simulators.

Debugging tools are included in HPC/VORX to help programmers determine errors that cause erroneous program behavior. VORX currently provides two debugging tools: a symbolic process debugger and a communications debugger. The communications debugger is used to isolate problems to a particular process, and the process debugger is used to find the particular lines of code that are in error.

Performance monitoring tools are used to find performance bottlenecks. In multiprocessor applications, a major bottleneck is one of load imbalance. If one processor has more work to do than others, the other processors waste time waiting for the slow processor to finish. VORX provides a performance monitoring tool that allows the programmer to see load imbalances graphically in real-time.

2. The Communications Debugger

2.1 Motivation

A common symptom of programming errors in multiprocessing applications is that the application deadlocks with each process waiting for input from other processes. Traditionally, programmers make use of a process debugger, like *sdb*[3] or *dbx*[4], that allows them to examine the program counter and variables of each process. When the

application consists of many processes, such debuggers are not very helpful because the programmer may have to separately examine the state of every process in the application in order to determine the programming error. Making matters worse, it may be that the state of each process looks reasonable when viewed in isolation and that the error can only inferred from the global state of the application.

Cdb is a tool to help debug such deadlocked applications. It is implemented on the HPC/VORX local area multicomputer system, but the ideas presented here should apply to any parallel computing system or local area network that uses message-based inter-process communications. *Cdb* allows the programmer to easily interrogate the communications state of an application. Since these deadlock problems are communications-related, the information presented by *cdb* can be used to determine which messages caused the problem and can often help isolate the process that caused the deadlock to occur.

When a program abnormally terminates in a uniprocessor system, say because of a bus error, the first thing that a programmer does is to invoke a process debugger and obtain a stack trace to determine where the error occurred in the program. Often this provides enough information to expose the bug. If not, the programmer proceeds to try other techniques such as examining variables or rerunning the program with breakpoints.

Like stack traces for uniprocessor applications that abnormally terminate, *cdb* is intended to be the tool that a programmer tries first for multiprocessor applications that deadlock. It should be easy to invoke, and a programmer should be able to interpret its output with no more than a few minutes of thought. Of course, *cdb* will not always suffice and the programmer may have to try other tools, such as program replay systems [5] [6] [7], program animators [8] [9] [10] [11], or painfully invoking a process debugger on each process of the application.

2.2 Cdb

Before describing *cdb,* we first quickly review the relevant properties of *channels,* an inter-process communications mechanism provided by VORX [12]. Channels are two-ended, named, communications objects that may be dynamically created and destroyed during program execution. They provide for the transmission of data from the address space of one process to the address space of another process and provide synchronization between readers and writers. Each channel has an arbitrary name, and two processes rendezvous on a channel by specifying its name in an open call. Data is sent on a channel with a *write* call and is read with a *read* call.

Since VORX channels are independent of each other, the communications state of an application is simply the union of the states of all its channels. The output of *cdb* thus consists of a display of the state of each channel. It displays the name of the channel, which two processes it connects, how many messages have been sent in each direction on the channel and most importantly, the state of each end of the channel. Because VORX provides blocking communications primitives, the state may indicate that a process is blocked waiting for input to arrive or for output to be sent. A complete list of the state values is given in Table 1. A valid state consists of zero or more of these values. Each VORX process may have many subprocesses (also known as threads). Since several subprocesses may be accessing a channel at once, it is possible for *cdb* to display states that would otherwise be impossible, such as both `READING` and `WRITING`.

OPENING	an open request has been requested for only one end of the channel
READING	the process is blocked reading the channel
READNING	the process is blocked reading the channel with a multiplexed read
SELECTING	the process is blocked at a multiplexed wait including this channel
INPENDING	input has arrived that has not yet been read by the process
n-IN_BUFFER	user buffering was specified, and the buffer contains *n* messages
WRITING	the process is blocked writing the channel
RCVDCLOSE	the process at the other end of the channel has issued a `close`
SENTRNR	data was received but all buffers were full

Table 1. Possible values for the state of a channel end

Because VORX allows a programmer to run several independent applications at the same time, each of which may encompass many processes running on many processing nodes, *cdb* must be able to determine which processes belong to the application of interest. It does so by presenting the programmer with a list of all the programmer's processes, giving the program name and arguments for each process. The programmer then chooses any one process that belongs to the application. *Cdb* performs the transitive closure of the channel connections from the specified process. Since all the processes of the application should be connected by channels, this operation finds all the processes comprising the application. Before displaying information about the application's channels, *cdb* displays a list of the processes that it has found. If this list does not contain all the processes in the application, the channels between processes must not be set up correctly, a common cause of deadlock.

A sample of the output from *cdb* is shown in Figure 2. In this example, the programmer typed *511* on the fourth line to indicate that process 511 was part of the application of interest. *Cdb* then displays the four processes of the application, preceded by their machine and process number. The output concludes with the channel display. The first channel listed has name `511 cntrl-0` and connects the process on machine 72 with that on machine 93. The process on machine 72 is blocked at a multiplexed read and has sent 869 messages on the channel. There is no current activity on the other machine for this channel, and 140 messages have already been sent.

While *cdb* is intended for use on applications that are deadlocked, it can also be used to spy on running applications. The information from each end of the channel is obtained as a slightly different time, so the states of the two ends may appear to be inconsistent with each other.

2.3 Wcdb

One of the problems with *cdb* is that applications with many processes tend to have far too many channels to fit on the screen. A simple way to handle this problem would be to place the output of *cdb* in a file and either peruse it with a text editor or send it to a printer. A better approach would be for *cdb* to provide mechanisms to zoom in on the channels of interest. These mechanisms are implemented in a window and mouse oriented version of *cdb* called *wcdb* that runs on the X window system [13].

```
host 0 (peri):
  511 vx emut 3
  522 vx -c3 engine 3 -1 511
Choose a process: 511

4 processes:
72/1: vx emut 3
93/1: vx -c3 engine 3 -1 511
91/1: vx -c3 engine 3 -1 511
75/1: vx -c3 engine 3 -1

9 channels:
511 cntrl-0 : 72/1 - 93/1   READNING (869) --- (140)
511 cntrl-1 : 72/1 - 91/1   READNING (875) --- WRITING INPENDING (139)
511 cntrl-2 : 72/1 - 75/1   READNING (881) --- (139)
511 cemu 0-0: 93/1 - 93/1   (464)          --- READNING (0)
511 cemu 0-1: 93/1 - 91/1   READNING (153) --- READNING (154)
511 cemu 0-2: 93/1 - 75/1   READNING (151) --- READNING (151)
511 cemu 1-1: 91/1 - 91/1   (472)          --- READNING (0)
511 cemu 1-2: 91/1 - 75/1   READNING (151) --- READNING (152)
511 cemu 2-2: 75/1 - 75/1   (464)          --- READNING (0)
```

Figure 2. Typical output from cdb

When *wcdb* is started, it produces a window that contains a listing of all the processes started by the programmer. The programmer then clicks the mouse on a process of the application of interest, and *wcdb* displays a list of the application's channels in the window as shown in Figure 3. If there are too many channels to fit, the first window's worth of the channels are initially displayed. The scroll bar to the left of the window is manipulated with the mouse to see the rest of them. While the channels are displayed, the programmer can click the mouse over one of the channels to see additional information, such as the contents of the last messages sent on the channel.

The top of the window has several buttons that are used to interact with *wcdb*. The `Filters` button is used to modify the channel display to help the programmer find channels of interest. When this button is clicked, a menu with the following choices appears:

```
Channels for One Process
Channels for All Processes
Sorted by State
Sorted by Channel Name
Sorted by Process
```

The first choice allows the programmer to view only channels with one end on a specified process. It causes a list of the processes of the application to be displayed. When the programmer clicks on one of these processes, the process display disappears and the channels to and from that process are displayed. The second choice causes the channel display to revert to the original list of all the channels for the application.

The three remaining choices cause the channel display to be sorted, allowing the programmer to scroll through the list, looking for patterns, or for channels with

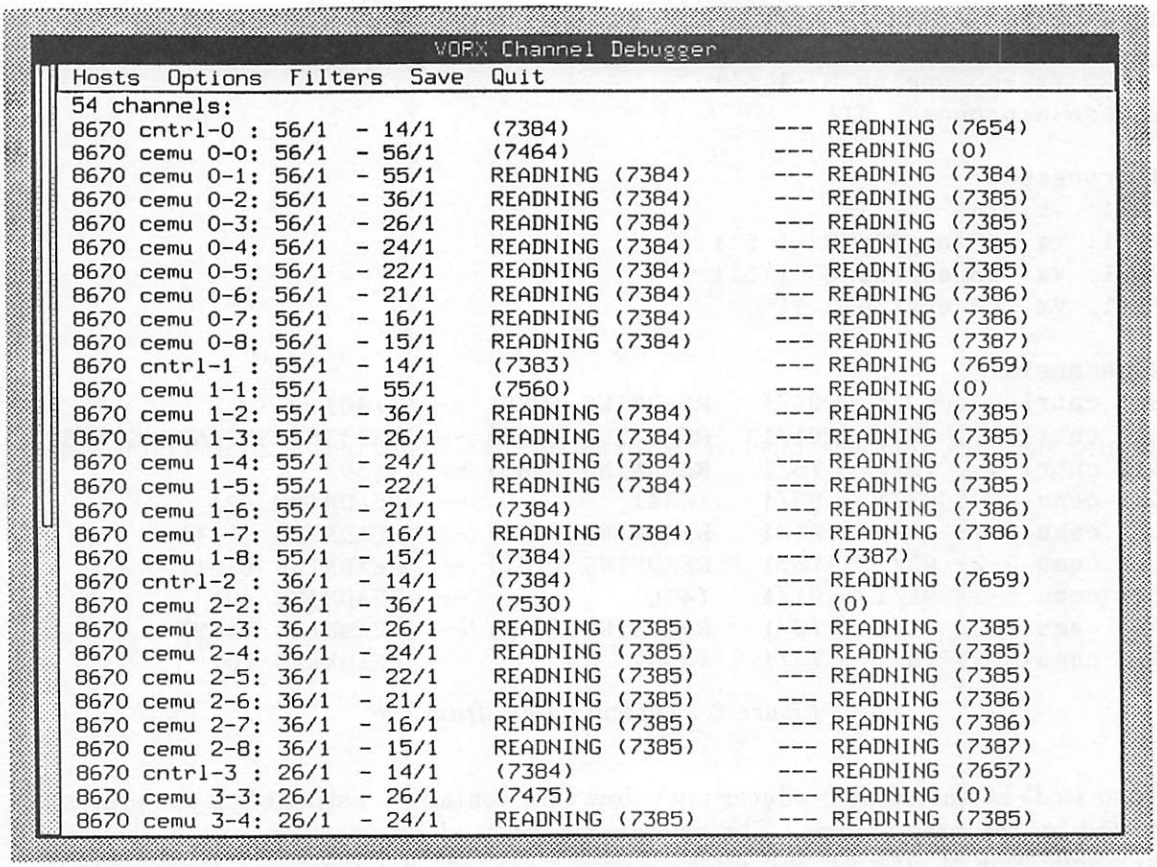


Figure 3. Typical output from wcdb

unusual states that look suspicious. Of the three sorts, the **sort by channel name** is the most useful, because channel names are usually chosen to have significance. The **sort by state** sorts by the state information displayed by *wcdb*. Because channels with similar states are grouped together, this display sometimes helps to find channels with unusual states. Of all the filters, the most popular is to only display the channels for a particular process, probably because the list of channels is usually short enough to all fit in the window at once.

The other buttons perform a variety of functions. The **Hosts** button is used to restart *wcdb*. When this button is clicked, a menu with the names of all the VORX hosts pops up. After a host is chosen from the menu, *wcdb* displays a list of processes on that host started by the programmer. Clicking on one of these processes chooses the application just like when *wcdb* is started. The **Display** button is used when the list of channels is being displayed. It pops up a menu that is used to switch the display back and forth between the list of channels and a list of the processes comprising the application. The **Save** button saves a copy of the current channel list in a file and the **Quit** button causes *wcdb* to exit.

2.4 Retrospective

Cdb and *wcdb* have proven to be popular debugging tools for VORX programmers. They have been most useful in the early stages of program development when the programmer is debugging the setup of communications between processors. It is not unusual for an application to deadlock the very first time that it is run. A common cause of this error is that some process does not use the correct name for a channel, so the rendezvous for opening the channel does not occur. This is immediately apparent from the output of *cdb* because the state of the channel is `OPENING`.

Another communications error that causes deadlock is for the programmer to not correctly match up message transmission from one end of a channel with message reception on the other. For channels used bidirectionally, this is usually manifested by a channel with the processes on both ends of the channel `READING` or both `WRITING`. Otherwise, any channel end that is blocked `WRITING` is suspicious, and finally channels that are blocked `READING` may be suspicious. Unfortunately, there are normally many channels in this last state, and most are a result of correct program execution.

For applications that deadlock after several seconds of execution, the message counts are sometimes useful. Often one channel will stand out as having one fewer message than the others like it, suggesting where to look for the program bug. While debugging an application, a programmer often makes hypotheses about the cause of program error. The information presented by *cdb* can often be used to help prove or disprove these hypotheses.

When a process exits, VORX automatically deallocates its end of a channel, causing the state of that channel end to become unavailable. If a processes abnormally terminates, say due to a bus error, VORX leaves the process in a dormant state with its channel connections still intact. This gives the programmer a chance to run *cdb* on the application. The state of its channels, and in particular the last messages sent to the errant process, are sometimes helpful in determining why the process terminated abnormally.

It is possible to apply *cdb* to a running application. Because *cdb* does not attempt to synchronize the data that it collects from the processors of an application, inconsistent information may be displayed. However, *cdb* has been used to monitor a running application to determine whether it has deadlocked. If the programmer knows that the application performs communications at some minimum rate, say once per second, then *cdb* is run twice, a second apart. If the message counts displayed by the two runs of *cdb* are different, then the application has not deadlocked.

2.5 Evaluation

Cdb is easier to implement than tools like program replay systems and program animators, yet is useful for finding many programming errors. Most of the information that *cdb* needs was already encoded in the communications driver, so the execution overhead associated with *cdb* is minimal. In VORX, the information used by *cdb* is always available: nothing needs to be done in advance to be able to invoke *cdb* on an application. It should be equally easy to implement a communications debugger like *cdb* on other systems with similar communications primitives, such as sockets in the BSD UNIX® System [14].

It has been useful to have both a line-oriented and window-oriented interface to the communications debugger. *Cdb* is usually used for a quick perusal of applications with few enough channels to fit on the screen and *wcdb* is used for more extensive debugging sessions. Unfortunately, not all of our users are familiar with the X window system, so the line-oriented *cdb* is the only one of these tools accessible to them.

There are many enhancements that we would like to make to *cdb*. One is a mechanism for *cdb* to find and display the set of channels causing the deadlock. When there are a large number of channels, it is tedious for the programmer to determine where the deadlock is, suggesting that it would be useful to automate this task. To provide this information, *cdb* would have to know the communications state of each channel relative to subprocesses, instead of processes as it is now does. It would probably be useful for *cdb* to be able to break down the communications by subprocess in any case.

It would be useful to integrate *cdb* with a process debugger. For instance, the programmer should be able to click on a channel in *wcdb* and have a stack trace of that process immediately displayed. This would make it easy to correlate the communications state with the state of a process. For more extensive debugging, there should be a way to invoke the process debugger directly from *wcdb*.

Another feature that might be useful would be to include an indication of the time that the last message was sent on each channel. With a scheme similar to that of TEMPO [15], it should be possible to synchronize all the clocks in the system within a millisecond or so.

3. Traditional debugging tools

Once a problem is isolated to a single process, the VORX symbolic debugger *vdb* can be used to find which part of the program is in error. *Vdb* is derived from the *sdb* [3] debugger. *Vdb* includes some features not included in *sdb*, such as the ability to switch between subprocesses for the purpose of examining their local variables. *Vdb* can be used to examine aborted processes, or can provide breakpoint debugging for one process while the other processes of the applications run normally. When used on a workstation with a window system, it is possible to do breakpoint debugging on a multiprocess application by starting several copies of *vdb* in separate windows. Each copy of *vdb* controls the execution of one process of the application. By switching between windows, the programmer can simultaneously debug all the processes.

When there are more than a few processes, this method becomes unwieldy because the programmer cannot remember what he is doing in each window. In practice, programmers usually run one or two processes with the debugger and run the other processors normally. Because the programmer may not know in advance which process needs to be debugged, VORX makes it possible for the programmer to attach *vdb* to any process that is running and to switch between the processes of his application. This feature is especially helpful for examining deadlocked programs. When a potential problem is identified with *cdb*, *vdb* can be attached to that process to find its exact state and determine whether it is the cause of the problem.

4. The Software Oscilloscope

4.1 Motivation

We are interested in developing tools to help a programmer improve the performance of an application running on a multiprocessor computing system. For single-processor computers there is a well developed technology for speeding up programs: a typical execution of the program is monitored by a profiling system that shows how execution time is divided among different parts of the program [16]. Typically one finds that a large proportion of the execution time spent in a small section of the program. This part of the program can then be examined and carefully rewritten.

We wish to monitor applications that run on a multiprocessor system with hundreds of processors that communicate by message-passing. These applications are more difficult to understand than those running on a single processor. The major problem is one of improper load balance: some processors spend time waiting for data from other processors instead of doing useful work. A related problem is that communication between processors is often more expensive than envisioned by the designers of an application, exacerbating the load balancing problem.

One approach to this problem is to collect information during program execution and to provide summary statistics after the program is run. This can be viewed as an extension of the profiling technique where information such as time spent waiting for other processors and communications costs have been added. However in some applications, such as circuit simulators [17], the characteristics of the computation change significantly during execution, resulting in a load balance that changes over time. Such applications require more information than is provided by summary techniques.

4.2 The Software Oscilloscope

We have designed and implemented a graphical display tool that runs on a color workstation and can display the execution characteristics of an application in real-time. The tool is called the *Software Oscilloscope* and was inspired in part by Sun Microsystem's perfmeter [18] program. It displays three synchronized sliding graph displays for each processor.

One graph indicates CPU time usage, with different colors used to partition time into several categories. Two of the categories are quite standard: *user* time in which application code is executed and *system* time in which operating system code is executed. The remainder of the time is *idle* time in which the processor is doing no useful work.

Because programs use messages to communicate, idle time can be further partitioned to provide more information. The processor may be idle because the program is waiting for input or it may be idle waiting for output. Because the kernel allows multiple threads of execution within a processor, a third possibility for idle time is that some threads are waiting for input and others are waiting for output. Finally, the processor may be idle for some other reason, such as waiting for access to a local disk.

The two other graphs indicate communications activity. One shows the number of messages sent (normalized to messages per second) and the other shows the number of messages received.

Because the graph of CPU time divides time into six different categories, and the time spent in each category can change radically in short periods of time, a color display is indispensable for presenting this data. However, a color display is not always available, so we sometimes run the Software Oscilloscope on a monochrome display. In this case, it only displays user and system time: idle time is not partitioned further. A typical monochrome display is reproduced in Figure 4.

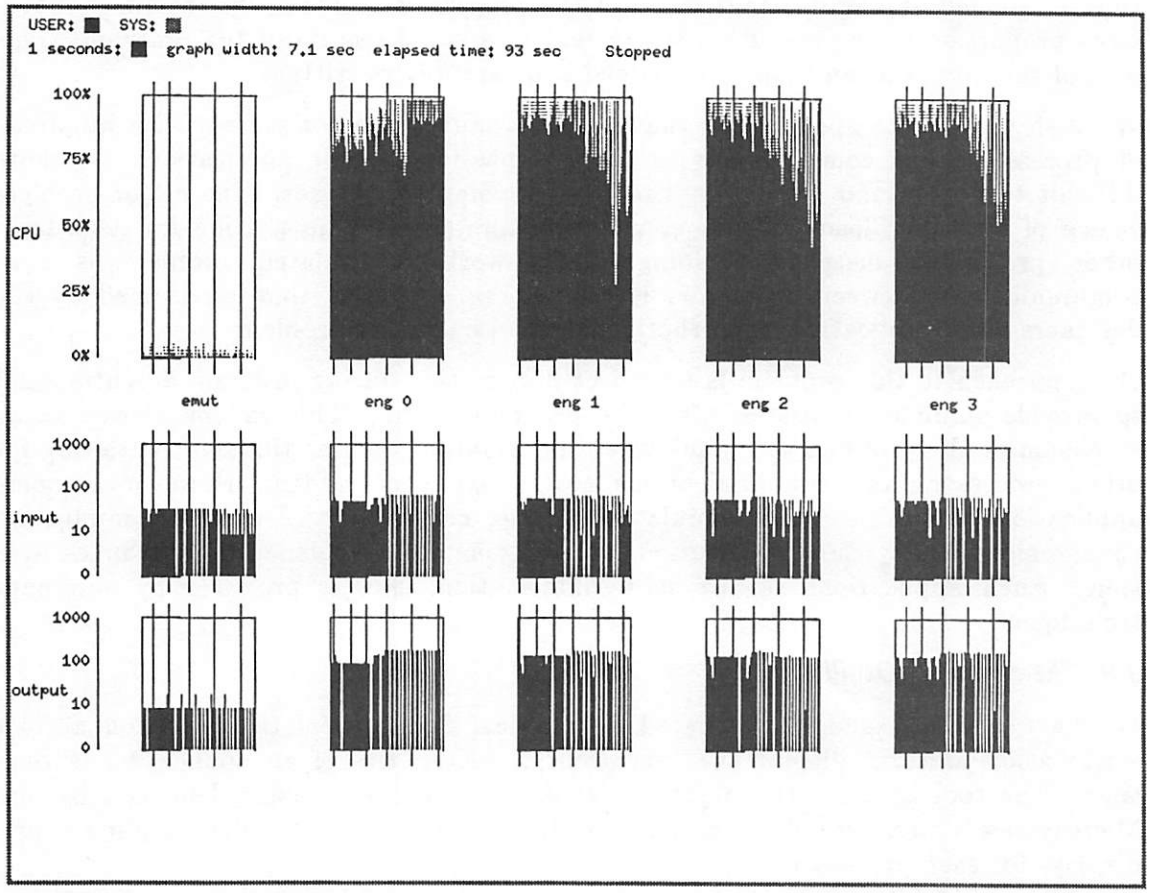


Figure 4. Monochrome approximation of typical output of the Software Oscilloscope

The monitor can be run in real-time mode with the display being updated while the application is running. While this is fun to watch, it is not very useful because information quickly slides off the graphs. We therefore also allow the display to be viewed off-line after program execution is finished. In off-line mode, the display may be frozen or run either faster or slower than real-time. It is also possible to seek to a particular moment in recorded time to review or skip parts of the display.

We have found it useful for applications to indicate moments of interest during execution by annotating the display with tick marks. For example, a simulator may insert a tick mark at each simulated time step. The tick marks and their labels slide along with the graphs allowing the viewer to correlate the display with program execution. These tick marks are visible as the long vertical lines in Figure 4.

Except for adding subroutine calls to insert tick marks, no source changes are required for a program's execution to be monitored. The current implementation requires that programs be compiled with an option that causes the monitoring system to be initialized before *main* is called. Profiling has a minor impact on the applications performance: it uses less than 5% of the processor's resources.

4.3 Current research topics

One interesting problem is to determine the appropriate resolutions for recording and displaying data. If it is too slow, then details of the execution may be missed. If it is too fast then the graph appears too jumpy and is hard for the viewer to understand. For the applications we have examined, a resolution between 50 and 500 milliseconds seems appropriate.

We are interested in using the Software Oscilloscope to monitor programs with hundreds of processors. It is clearly infeasible to display hundreds of tiny graphs on the display. To solve this problem, we are devising an intelligent interface that automatically picks out interesting processors and displays only these. A serious problem here is dealing with the large amount of data coming from the hundreds of processors.

We plan to incorporate real-time profiling capabilities into the Software Oscilloscope by subdividing the display of user time. When program execution is started, the names of routines to be profiled would be specified, and each would be displayed in a different color within the CPU time graph.

4.4 Observations

We have used the Software Oscilloscope to monitor a version of the CEMU circuit simulator [17]. As noted earlier, we observed that the load imbalance does change over time, but were surprised to find that the interval between these changes is often several seconds. This suggests that execution time could be improved by monitoring the load balance in the application and periodically moving regions between processors to make the load more even.

The Software Oscilloscope appears to be a useful tool for monitoring multiprocessor programs. The current version works well for monitoring up to about ten processors and assumes that a single process runs on each processor. Work remains to be done to remove these limitations.

5. Conclusions

We have been surprised at how often *cdb* is used to debug programs. The combination of *cdb* and *vdb* has proven to be sufficient to deal with most program bugs. For bugs, such as timing-related problems, that these tools are not capable of handling, VORX provides a logging mechanism that allows programmers to write debugging output into a large kernel buffer while the program runs and to examine the buffer after the program terminates. So far, we have had no demand for more exotic tools such as replay systems or animators.

On the other hand, the Software Oscilloscope has received less use than we expected. A primary cause is that most programmers are interested in examining programs with more that ten processors. We expect that the Software Oscilloscope will become more popular when it is extended to deal with a larger number of processors.

REFERENCES

1. Gaglianello, R. D., et al., "HPC/VORX: A Local Area Multicomputer System," *Proc. Ninth Internat. Conf. on Distr. Comput. Sys.*, June 1989, Newport Beach.

2. Ensor, J. R., et al., "The Rapport Multimedia Conferencing System—A Software Overview," *Proc. Second IEEE Conf. on Comput. Workstations,* Washington, March 1988, 52-58.

3. Katseff, H. P., "Sdb: A Symbolic Debugger—Version 3.0," part of documentation for the UNIX® System V Operating System, AT&T, 1980. Also AT&T Bell Laboratories Technical Memorandum 80-1353-12.

4. 'DBX (1),' in *UNIX Programmer's Manual,* 4.2 Berkeley System Distribution **1,** Computer Science Division, University of California, Berkeley, CA, August 1983.

5. Feldman, S. I. and C. B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 112-123.

6. Pan, D. Z., and Linton, M. A., "Supporting Reverse Execution of Parallel Programs," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 124-129.

7. Forin, A., "Debugging of Heterogeneous Parallel Systems," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 130-140.

8. Hough, A. A., and Cuny, J. E., "Initial Experiences with a Pattern-oriented Parallel Debugger," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 195-205.

9. Socha, D., et al., "Voyeur: Graphical Views of Parallel Programs," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 206-215.

10. Rubin, R. V., et al., "Debugging Parallel Programs in Parallel," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 216-225.

11. Stone, J. M., "A Graphical Representation of Concurrent Processes," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging,* Madison, May 1988, 216-235.

12. Gaglianello, R. D. and Katseff, H. P., "Communications in Meglos," *Software Practice and Experience* **16**(10), October 1986, 945-963.

13. Scheifler, R. W., and Gettys, J., "The X Window System," *ACM Trans. on Graphics* **5**,2, April 1986, 79-109.

14. 'SOCKET (2),' in *UNIX Programmer's Manual,* 4.2 Berkeley System Distribution **1,** Computer Science Division, University of California, Berkeley, CA, August 1983.

15. Gusella, R., and Zatti, S., "TEMPO—A Network Time Controller for a Distributed Berkeley UNIX System," *1984 Summer Conference Proceedings,* USENIX Association, Salt Lake City, 1984.

16. Graham, S. L., Kessler, P. B., McKusick M. K., "gprof: a Call Graph Execution Profiler," *Proc. ACM SIGPLAN '82 Symp. on Compiler Construction,* SIGPLAN Notices **17**, 6, June 1982, 120-126.

17. Ackland, B. D., et al., "MOS Timing Simulation on a Message Based Multiprocessor," *1986 IEEE International Conference on Computer Design: VLSI in Computers and Processors,* Port Chester, New York, October, 1986.

18. "Perfmeter," *Sun Microsystems Command Reference Manual,* Part No: 800-1295-02, Sun Microsystems, Inc., Mountain View, CA, 1986.

CAPS - A Coding Aid used with the PASM Parallel Processing System

James E. Lumpp, Jr., Samuel A. Fineberg†, Wayne G. Nation,
Thomas L. Casavant†, Edward C. Bronson, Howard Jay Siegel,
Pierre H. Pero, Thomas Schwederski‡, Dan C. Marinescu*

Parallel Processing Laboratory, School of Electrical Engineering
* Department of Computer Science
Purdue University
West Lafayette, IN 47907 USA

ABSTRACT

A major difficulty with programming parallel processing systems is the debugging phase of software development. The ability to monitor code on conventional non-parallel machines through such simple features as breakpoints and print statements is invaluable in debugging serial code. However, debugging code on parallel machines with these same features is hampered by the lack of support for I/O from the individual nodes of the parallel system. This work presents the implementation of a remote execution and debugging/monitoring environment for the PASM prototype, a 30-processor parallel machine designed and constructed at Purdue. CAPS (Coding Aid for the PASM System) presents information from the individual nodes of the system to the user on a high-resolution workstation. CAPS is currently used to support development of software tools, languages, and applications. CAPS is described along with several design alternatives that were considered. Its scalability to larger systems and future support are also discussed.

This work supported by the National Science Foundation under Grant Numbers CCR-8702115, CCR-8704826, CCR-8809600, and CCR-8846388, by the NSF Software Engineering Research Center (SERC), by the SDI under ARO contract DDAAL03-86K-0106, and by the Purdue University School of Electrical Engineering.

† Current address: Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242.

‡ Current address: Institute for Microelectronics Stuttgart, Allmandring 30a, D-7000 Stuttgart 80, West Germany.

1. Introduction

Programming parallel machines is very difficult. First, generating an algorithm requires the programmer to assimilate multiple threads of control. Second, synchronization and communication between the threads must be addressed to avoid contention and deadlock. Then, once the program is executing on the parallel system and does not function correctly or performs poorly, the debugging* of multiple threads is a complicated problem. The programmer requires information about the run-time characteristics of the program in order to correct its operation or optimize its behavior. Unfortunately, this information is very burdensome to obtain for a parallel system and few existing parallel systems include sufficient architectural support to provide it.

CAPS integrates hardware support and software tools to provide a remote execution and program debugging/monitoring environment for *PASM*, a partitionable SIMD/MIMD parallel processor prototype designed and constructed at Purdue University. CAPS (Coding Aid for the PASM System) is the current generation of monitoring hardware and software for PASM. This includes specialized hardware added to the prototype and software servers, running on PASM and the user's workstation, to facilitate the transfer of information. CAPS is currently used to assist development of application and system software for PASM, as well as in experimental system evaluation [BrC89a, BrC89b, FiC88a]. An environment of this type is useful for several reasons. First, it allows the machine to be accessed from a remote site. The programmer need not literally stand in front of the machine to use it. Second, on a partitionable machine, multi-user access is permitted with users working at separate remote sites using different parts of the machine concurrently. Downloading application code and the development of code may be integrated into the same remote environment. The remote machine used for program development may have software tools not available on the target machine. Third, support for run-time monitoring of parallel programs is crucial in the debugging phase of parallel software development. Information gained through the actual execution of the program on the parallel machine is invaluable in correcting and optimizing the operation of programs. Finally, the addition of such an environment to an existing machine can be inexpensive, as shown by the implementation presented here.

CAPS consists of a set of dedicated I/O channels and associated hardware and software that facilitate bi-directional information flow between the individual nodes of PASM and a workstation providing the user interface. The information is sent by code added to the user's program that transmits messages through the dedicated I/O channels. Once the data is sent from the nodes it is combined into a single stream that is sent through a high bandwidth *Local Area Network* (*LAN*) to the workstation where it is used to debug and analyze the execution of programs. Currently, only ASCII data is sent and the information is presented to the user in a textual form. However, work is underway to develop useful

* In this paper *debugging* refers to the process of modifying the program to execute both correctly and efficiently, e.g., the identification and elimination of contention is considered debugging.

graphical user interfaces.

The monitoring data is sent by code added to the user's program, so the natural (un-monitored) flow of the computation is altered. Because parallel programs often depend on the synchronization of threads of control (e.g., the order of access to shared resources is important), the change in flow of the computation can greatly affect the execution of the program. This phenomenon is called *intrusion* by the monitoring system.

For serial programs, debugging information is relatively easy to obtain and can be gained solely with software probes. Because there is a single thread of control, this thread can usually be arbitrarily delayed while the state of the program is observed. Breakpoints can be set and the contents of all program variables checked in order to obtain information about the instantaneous state of the program. Utilities exist that aid the programmer in setting breakpoints and for examining the state of the program. Tools are also available for profiling serial programs to obtain statistics on the execution time of various sections of the program. These techniques are invaluable in the debugging phase of programming for serial computers that often consumes the bulk of the program development time.

These techniques, however, cannot be directly extended to the parallel case due to the existence of multiple threads of control in parallel programs [MiC88]. It becomes necessary to provide some architectural support in the form of hardware instrumentation to aid debugging efforts. A software probe of one thread can only supply local information and cannot affect other threads (e.g., stop them in order to examine system state). In addition, if the monitoring system affects the relative timing of different threads, the execution time of the program may increase, or (worse yet) deadlock situations could be created or masked and the debugging of the parallel program may become impossible. As an example, consider the debugging of a program in which the ordering of two events in different nodes is important. If, depending on the input data, the monitoring sometimes changes the ordering of these two events then debugging becomes impossible.

Some degree of hardware instrumentation is necessary to keep intrusion to a reasonable level. CAPS was designed to keep the hardware enhancements to the architecture of minimal cost. The techniques used can be applied to a broad class of parallel machines that includes the PASM parallel processing system, and any system capable of executing in either SIMD mode, MIMD mode, or both. The individual processors of these machines must have the capability to transmit debug/trace information over an I/O channel to a location where the information can be collected and forwarded to a remote site.

Section 2 provides background information about environments to support software development. A brief overview of the PASM parallel processing system prototype on which CAPS was developed is given in Section 3. Section 4 describes the user's view of CAPS. The design alternatives considered and the architectural enhancements made to support the CAPS system are discussed in Sections 5 and 6. Section 7 presents next generation plans for debugging and

monitoring environments for systems such as PASM.

2. Background

A programming environment is designed to simplify the task of program development [Cha86]. Although what constitutes a programming environment depends on the designer's view of the key difficulties in the software development process, it is clear that there exist two distinct categories for the support these environments provide: support for the programmer to develop the program (*static support*) and support for the programmer in debugging the program while it is executing (*dynamic support*). Both static and dynamic support are important in the development of parallel programs, but dynamic support is crucial. Static support can aid the programmer to code the program more efficiently or make the task of programming easier, but dynamic support provides information on the actual interaction of the program and the parallel machine. This allows the programmer to optimize the program to a greater degree and aids in locating errors. Also, interactive I/O with the nodes of the system allows their individual states to be examined. CAPS provides dynamic support and interactive I/O for the PASM parallel processing system.

A number of static support tools aid the development of large software systems by extending the concepts of modular program design. Examples of systems motivated by this principle include Cedar [Tei84], Mesa [Swe85], Jasmine [MaW86], and Starlite [CoA86]. Additional benefits have been gained by applying knowledge-based program transformation (e.g., PDS [Che84], CHI [SmK85]) or by providing intelligent programmer assistance as in Programmer's Apprentice [Wat85]. Other static support environments are intended to increase the use of parallelism (e.g., PTOOL [AlB86], Poker [SnS86], and Pisces [Pra85]). More complex systems that augment a higher level parallel programming language with tools to aid the programmer in visualizing program behavior have also been proposed (e.g., PIE [SeR85] and CSP [DeS86]).

A number of systems also provide dynamic support. As early as 1975, McDaniel proposed a kernel instrumentation for distributed environments [McD75]. The work on some of the distributed systems designed in the early 1980's include some sort of performance monitoring facility, usually a software monitor (e.g., [ChZ85] for the V kernel and [PoM83] for DEMOS/MP). Systems for interactive debugging of a distributed computational environment were also designed [Sch81].

Currently, several systems are in different stages of completion. Among these are CARAT at the University of Massachusetts and the Distributed Computer Testbed at Honeywell. The Real-Time monitoring systems at Ohio State is based on an Ethernet or Hyperchannel [Van87]. IPS at the University of Wisconsin aids in guiding the user to the sources of inefficiencies [MiY87]. REMS (resource monitoring system) at NBS and the performance monitoring facilities for RP3 [BrM89] feature substantial hardware support. The Faust project for Cedar at the University of Illinois includes hardware and operating system support for time-stamping significant events occurring during program execution.

Instant Replay at CMU allows the replay of a program from trace files [LeM87]. The High Level Debugging environment at the University of Massachusetts is a sophisticated environment based on the EBBA paradigm (Event Based Behavioral Abstraction) [Bat88, BaW83]. Parasight at Encore is an example of a software monitor for shared-memory parallel processing systems [ArG88]. Also, the SEECUBE package allows the visualization of communication in parallel programs on a hypercube [Cou87].

Several systems have also been designed to provide interactive I/O with the nodes of the parallel machine. The examples listed below are representative of those available matching the class of machines at the focus of this paper.

The most common method of interactive I/O among the "global system bus" machines (e.g., Sequent Balance or ELXSI [Tay83]) is by means of a separate I/O unit on the system bus. This I/O unit handles all user I/O from the CPUs to a number of terminals connected to the system. All I/O passes over the system bus between the CPUs and I/O unit, increasing bus traffic and interfering with the computation unit. The FLEX is an example of a distributed memory "system bus" machine which can be configured with an I/O unit for each CPU [Fle85]. However, no device for concentrating data into a single stream exists. The BIOlink of a BBN Butterfly processing node can interface to an external device [ThG86]. But again, no instrumentation is included to combine user I/O from all processing nodes. The NCube machine has an internal channel permitting a processing node to send and receive information to/from the system's host processor [HaM86]. This information passes through separate I/O processing nodes. These systems provide user-directed I/O with either a single processor or a group of processors through dedicated I/O units. What is not supported is a coherent user environment that combines code development tools, graphics, and interactive I/O with all processors on a sophisticated workstation.

3. PASM Overview

PASM is a partitionable SIMD/MIMD machine being designed to include over a thousand processors [SiS87]. It is a dynamically reconfigurable architecture, where the processors can be partitioned to form independent virtual machines of various sizes. Each virtual machine can independently switch between the SIMD and MIMD modes of parallelism at instruction level granularity with negligible overhead (this is referred to as *mixed-mode parallel computation* [FiC88b]). In addition, a flexible multistage network is used for inter-processor communications. A 30-processor prototype has been constructed and is in use in the Parallel Processing Laboratory at Purdue's School of Electrical Engineering. A block diagram of the basic components of a PASM system is shown in Figure 1.

The *Parallel Computation Unit (PCU)* contains $N=2^n$ PEs (numbered from 0 to N−1) connected by an Extra Stage Cube interconnection network [Sie85]. Each *PE* (processing element) is a processor/memory pair. The PE processors are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The PE memory modules are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro*

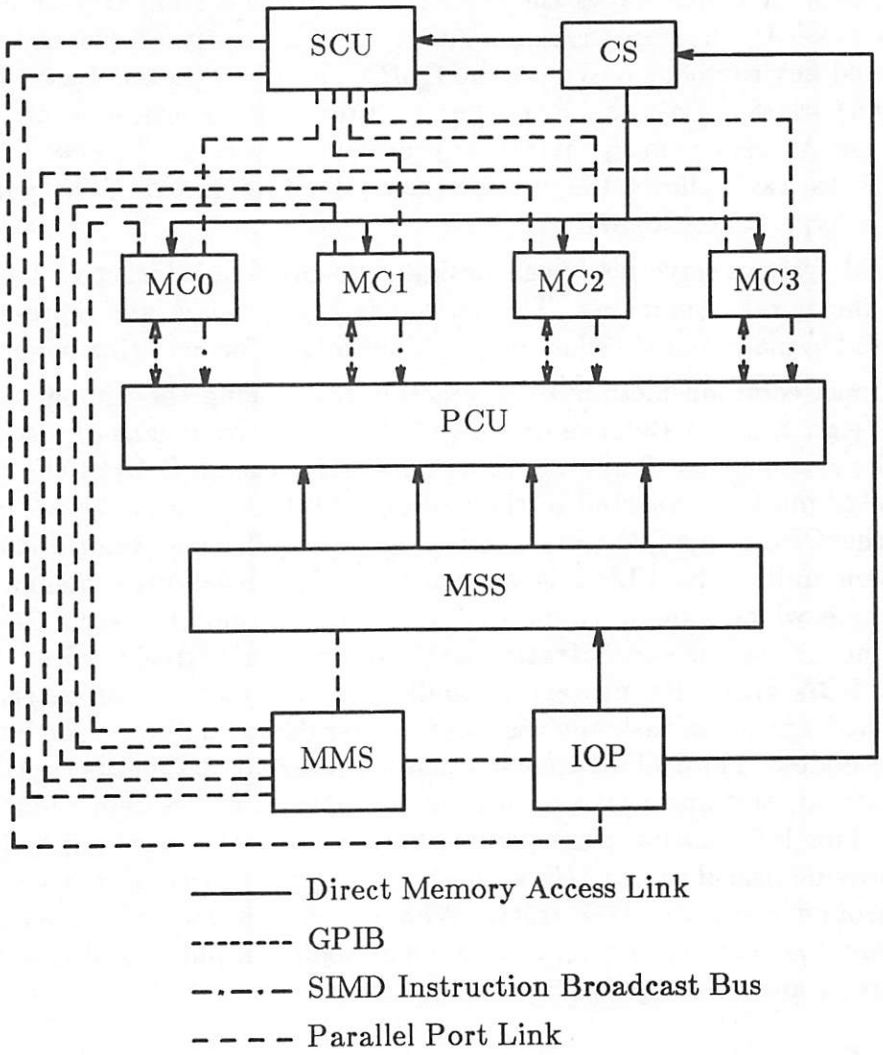


Figure 1. Block diagram of the PASM parallel processing system.

Controllers (MCs) are a set of $Q=2^q$ processors numbered from 0 to Q−1. The MCs act as the control units for the PEs in SIMD mode, sending instructions via the SIMD instruction broadcast bus, and coordinate the activities of the PEs in MIMD mode, through a General Purpose Interface Bus (GPIB). Each MC controls N/Q PEs. PASM is being designed for N=1024 and Q=32. The prototype has N=16 PEs and Q=4 MCs.

The *System Control Unit (SCU)* is responsible for the overall coordination of the other components of PASM. In the prototype, the System Control Unit connects to an Ethernet based LAN shared by several dozen minis and super-minis and over a hundred Sun workstations* on the Engineering Computer Network (ECN) at Purdue University. The *Memory Management System* (MMS) controls secondary storage and file transfer to/from the PEs. The *Memory Storage System*

* Sun Workstation is a registered trademark of Sun Microsystems Inc.

(MSS) provides multiple secondary storage devices for the PEs. The *Control Storage* (CS) provides secondary storage space for the MCs and the System Control Unit. One of the Memory Management System processors of the prototype, the *I/O Processor (IOP)*, is responsible for interfacing to external I/O devices and distributing data to the Memory Storage System. The System Control Unit communicates with the MCs and the I/O Processor through individual parallel port connections.

As previously mentioned, there are a total of 30 processors in the PASM prototype: 16 PE CPUs, four MC CPUs, four Memory Management System CPUs, one CPU associated with each of the four Memory Storage System disks and the Control Store disk, and the System Control Unit CPU.

4. User's View

The process of monitoring a program's execution begins with adding monitoring code statements to the user's source code. These statements send messages through a dedicated I/O channel to the monitoring system. The code to send the messages may be added either manually by the programmer to gain information on specific aspects of the program or by the system to obtain information on more global issues relating to program execution. The data sent from the nodes is collected and can be presented to the user in a number of forms including textual or graphical. The display can provide information on each node or on the system as a whole. The physical interface to the user is through a high-resolution graphics workstation running an interactive terminal screen control program called X-windows [ScG86].

To monitor the program, the user must decide which events or sections of the program are of interest and insert code that will mark their occurrence. When executing these sections of the program or events, the code inserted will mark the event according to its type and may record the time at which the event occurs for later analysis. The system may also mark some events of interest about the overall characteristics of the program, e.g., overall execution time. In this case the operating system will automatically mark sections of the code. Both approaches are *intrusive* measures because computation is interrupted for the time period it takes to mark the event. Currently in CAPS, all the debugging and monitoring code is inserted manually by the programmer in the form of macros that are expanded by the system.

The *user-intrusive approach* requires the most of the programmer but is quite flexible. It can be as simple as print statements scattered throughout the code. It can also be used to incorporate sophisticated logging operations with automatic occurrence time recording added to the user program. This log of events can be stored in memory and reviewed on the workstation once program execution has completed. At the workstation, data records from all PEs can be analyzed in either their raw form by the user or processed by the workstation to present a graphic timing diagram of the program execution or other meaningful displays. With this information the programmer can detect errors, bottlenecks in the code, network conflicts, etc., and make appropriate code modifications.

The *system-intrusive* approach releases the programmer from the chore of inserting monitoring probes, but unfortunately at the cost of some flexibility. The user is limited to studying only the events recognized by the operating system. Events such as system calls, network accesses, SIMD/MIMD mode switches, etc., can be logged and analyzed as described above.

A major advantage of a system like CAPS is the ability of the programmer to utilize some of the sophisticated features of a graphics workstation on I/O coming from the parallel system. The workstation's windowing capability allows the debugging information to be displayed as several interactive virtual terminals, each connected to a different processor, or to generate graphics displays summarizing collected data.

The graphics workstation in the current CAPS implementation is a Sun 3 [Sun86] running X-windows. These windows allow interactive I/O between the workstation and any processor on the PASM system. Each PASM processor can have its own window on the workstation. Each window can be adjusted to any size and windows may overlap if necessary. A window can act as a terminal allowing access to the processor's resident monitors, that are based on Motorola's MVMEBUG software [Mot83]. Monitor features include printing memory and register contents, setting program breakpoints, disassembly of segments of memory, and other debugging functions. The programmer may also send displayed information to disk or review information that has scrolled past the screen using standard X-windows utilities.

In a typical debugging session, the CAPS server on the Sun opens windows to PASM's System Control Unit, the host where the parallel program to be executed is being developed (usually a dual processor Vax 11/780 [GoM82]), and to the processors of interest on the PASM system. The window to the host machine is opened through a standard Unix* remote login procedure. CAPS is invoked on the Sun Workstation and opens windows to the desired PASM processors. The System Control Unit window can be opened via a remote login from the Sun or it can be opened through CAPS and provides access to Unix System V on the System Control Unit. In this configuration, the programmer is able to execute programs on PASM and use program output to help debug the software. The PE's resident monitors can also help detect errors. The monitoring information can be used to make changes in the source code on the host, then the program can be re-compiled/assembled, re-loaded, and re-executed on PASM, all without leaving the CAPS environment. This procedure can be carried out on any workstation capable of running X-windows attached to the LAN, as well as remotely from any Arpanet site with the same capabilities.

The sample screen of a Sun Workstation during a typical programming and debugging session is shown in Figure 2. Five windows are opened to a four PE machine partition consisting of MC 2 and PEs 2, 6, 10, and 14. This partition is running a mixed-mode SIMD/MIMD 8-point fast Fourier transform program. The window entitled "PASM Parallel Processing System" is open to the System

* Unix is a trademark of AT&T Bell Laboratories.

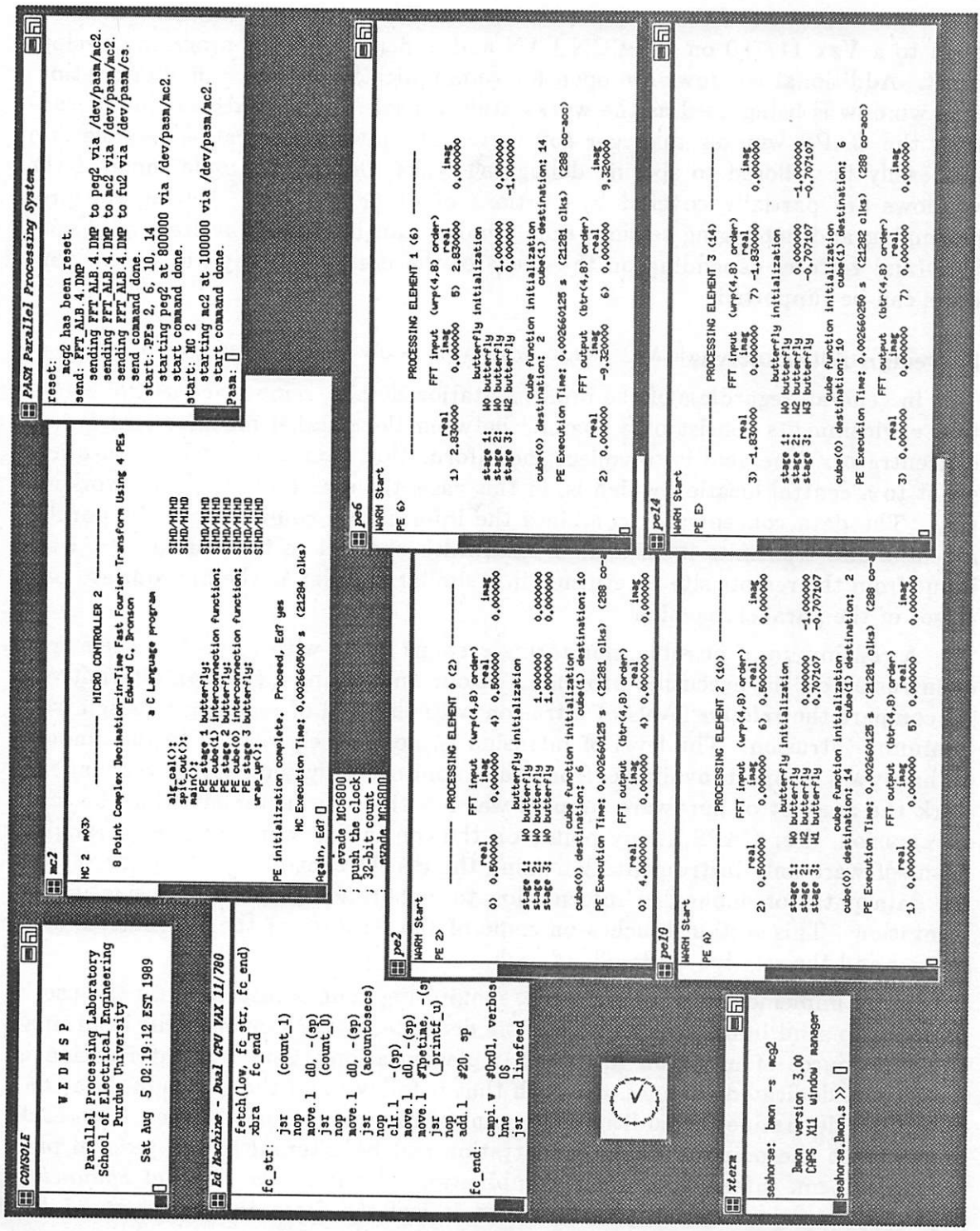


Figure 2. The screen of the Sun Workstation showing 9 windows: five opened to a machine partition with one MC and four PEs, one opened to the System Control Unit, one opened to a Vax and two opened to the Sun Workstation.

Control Unit and displays the operations necessary to download the program code and begin execution of the program on the four PE partition. Another window is open to a Vax 11/780 on the ECN LAN and is being used for program development. Additional windows are open for communication to the Sun Workstation. One window is being used as the workstation console and another window is running the CAPS window manager software. The interface is rather flexible and can easily be tailored to specific debugging tasks. In this example some of the windows are partially covered by portions of other windows. During the programming and debugging session, each window can be moved, resized, and iconized (and restore) depending on the needs of the user. Also, any number of windows can be supported.

5. Design Alternatives

In general, regardless of the implementation details, remote access and execution environments consist of a channel between the parallel machine and a data concentrator. The goal is to collect the information from each of the nodes and get it to a central location which is, in this case, the screen of the user's workstation. The data concentrator combines the information coming from the parallel machine and transmits it on a high bandwidth channel to the remote site. Data input from the remote site is returned in a similar manner to the appropriate processor of the parallel machine.

A continuum of possible monitoring systems exists with respect to their level of intrusion on the executing program. In our analysis of the design alternatives, we compare the *relative* levels of intrusion with the goal of reaching a design with minimum intrusion. The level of intrusion is most closely related to the amount of hardware support available from the monitoring system. However, in this work the amount of hardware support was not the only factor affecting the level of intrusion. For CAPS, many points on the continuum were considered ranging from software only instrumentation using the PASM control hierarchy [ScN87] as the data path for debugging information to elaborate hardware intensive implementations. This section touches on some of the features of the alternatives considered and the relative trade-offs of each.

In all implementations considered, monitoring code is inserted into the user's program to send information through the dedicated I/O channels. This is an unavoidable source of intrusion for these implementations. Once the information is sent to the dedicated channel, the path that it follows and the means of data concentration determine the additional amount of intrusion. The channels from each processor to the point of data concentration can be external to the parallel processing system, internal, or some combination of both. An *external channel* is physically attached to the processor and is isolated from the hardware of the parallel machine except for the single connection at each processor. An *internal channel* transmits information through data paths already embedded within the architecture of the parallel machine. A hybrid channel may use some internal channels as well as external channels to form the path that routes the information to a remote site. Ideally, the data paths used by the debug/trace statements are isolated from the rest of the parallel machine. This way, no part of the

computational hardware is affected by the overhead of the monitoring environment. This form of intrusion can be avoided by intelligent decisions about the placement of the monitor hardware support.

One hardware solution considered (*Annex approach*) uses an N-to-1 "concentrator" component that combines N serial ports connected to PASM's Parallel Computation Unit into a serial stream of data received and transmitted over a LAN. This hardware acts as a distributor of data for input from the windows of the workstation to the processors of the Parallel Computation Unit and as a concentrator of data from the Parallel Computation Unit processors back to the workstation. This single piece of hardware (manufactured by the Encore Computer Corporation under the name Annex^{TM}-UX terminal server [Enc86]) performs all of these functions. This design has the lowest degree of intrusion since no part of the PASM control hierarchy is involved. The only intrusion comes from the statements added to the user program to mark events. This solution was not chosen, however, because initial studies indicated that an alternative system (eventually chosen) would perform well and could be put into operation in a matter of weeks for a fraction of the cost of the Encore system.

At the other end of the continuum was an approach which required no additional hardware. This *embedded approach* uses the parallel I/O capabilities of the control hierarchy of PASM itself. In this solution, the parallel data paths from PE to MC and from MC to System Control Unit shuttle data packets between the Parallel Computation Unit and the System Control Unit. From the System Control Unit the Ethernet channel is accessible to send the packets to the remote site. This approach had the highest amount of intrusion with parallel computation because the monitoring/debugging information is passed along the same path as program control information interfering not only with the flow information but also incurring more overhead for the MCs to transfer the information to the System Control Unit.

The implementation chosen was a hybrid of the Annex approach and the embedded approach using the PASM hierarchy. Because the System Control Unit does not take part in the actual execution of the parallel program no intrusion occurs from the use of its Ethernet channel. Also, the path chosen between the PEs and the System Control Unit does not include any paths dedicated to parallel control. A new board, the *System Monitoring Module (SMM)*, was added to the backplane of the I/O Processor. The System Monitoring Module is capable of combining the signals from the Parallel Computation Unit and forwarding them to the I/O Processor. Because the I/O Processor is not a part of the Parallel Computation Unit it can also serve as an I/O channel without added intrusion. From the I/O Processor the data is passed to the System Control Unit without using any paths dedicated to the Parallel Computation Unit. Once received by the System Control Unit, the information is sent over a LAN to the monitoring workstation. The operation of the System Monitoring Module approach chosen will be discussed in detail in Section 6.

Another important consideration in the design was its scalability and ultimate limitations. When combining debug/trace information from N processors,

where N is arbitrarily large, there is ultimately some value of N that overwhelms the bandwidth of the system. Even if all issues of hardware and software scalability are overcome and the environment is capable of handling an arbitrarily large number of processors, the application programmer may not be able to utilize all the information provided. It is impractical to expect a user to assimilate information from 1024 processors simultaneously. Therefore, it is reasonable to consider alleviating the bottleneck through intelligent use rather than additional hardware.

Each of the design alternatives considered degrades in a different manner as the Parallel Computation Unit becomes larger. The embedded software approach begins degradation immediately with both the MCs and System Control Unit providing possible bottlenecks for the flow of data. The amount of intrusion also rises because the MCs take part in the parallel program's execution and are further burdened by the monitoring support they provide. The level of intrusion of the hardware solutions, however, is not affected by scaling of the Parallel Computation Unit, because the data paths for these approaches do not include any paths used by the Parallel Computation Unit.

As the size of the Parallel Computation Unit increases, the Annex solution would only require additional terminal servers each with independent connections to the LAN. In this case the only bottleneck possible would be the LAN or the workstation that would receive the data. The scaling limitations of the System Monitoring Module approach, as well as considerations to scaling to extremely large numbers of processors will be discussed further in Section 6.

The most efficient way to avoid bottlenecks in the system is through informed usage based upon observations of the properties of parallel programs and the process of debugging these programs. Consider some general characteristics of parallel programs and their programmers. Most parallel programmers initially write applications for a subset (partition) of the available processors and/or a reduced data set, scaling their program and/or data after debugging is finished. By writing programs that can execute on partitions of varying size, the programmer gains two advantages. First, the programmer can debug and test code on a small number of processors. Second, the program will run on whatever size partition is available at a later time. The partition size may be determined by the user's data set or machine usage at run time. Also, some parallel programs have only a small number of unique processes distributed as identical copies on a large number of processors. When debugging such programs, the programmer can choose a representative set of the processors to monitor initially. As testing continues, this set of monitored processors can change depending on errors encountered in the code or on the events the programmer wishes to monitor. With these considerations it should be possible to limit the number of processors that must be monitored to a number manageable to both the environment and the programmer. Exceptions include some performance measures where contention grows non-linearly with system size.

6. Architectural Support for CAPS

Each of the CPU boards in the PASM prototype has a serial port intended for terminal I/O with the CPU's resident monitor to allow for program debugging and control. CAPS uses these I/O channels. Each serial port in the prototype is connected to the System Monitoring Module, which is controlled by the I/O Processor. The System Monitoring Module and I/O Processor together act as the data concentrator.

A block diagram showing how the architectural support for CAPS is integrated into PASM is shown in Figure 3. The CAPS system on the PASM prototype functions in the following manner. The I/O Processor constantly monitors each of the serial ports of the System Monitoring Module for incoming data from any of the PASM CPUs. Once a PASM CPU sends a character out its own serial port, the associated port on the System Monitoring Module receives the character and stores the character. The I/O Processor reads the PASM CPU's transmitted character and forms a two-byte packet. The first byte of the packet contains information indicating which of the PASM CPUs sent the character. The second byte of the packet is the 7-bit ASCII character sent. The I/O Processor sends this packet to the System Control Unit via the I/O Processor - System Control Unit parallel port connection. A process running on the System Control Unit reads the packets from its parallel port connection and sends the packets out onto the Ethernet channel to the Sun Workstation. Data input through the windows on the Sun are packetized and returned to the appropriate PASM CPU in a similar manner, i.e., Sun to System Control Unit, System Control Unit to I/O Processor, I/O Processor to System Monitoring Module, System Monitoring Module to PASM CPU.

The data concentrator (System Monitoring Module - I/O Processor pair) is necessary because no other component of PASM, e.g., System Control Unit or I/O Processor, has the number of ports required to bring all CPU serial connections together. The I/O Processor controls the System Monitoring Module rather than the System Control Unit because the ports must be serviced in real-time to avoid loss of data. The System Control Unit, running Unix V, is not able to service that number of ports without neglecting its other activities or losing data. However, the System Control Unit is capable of handling the single stream of packetized data from the I/O Processor. When the System Control Unit is unable to service the I/O Processor - System Control Unit parallel port, the I/O Processor buffers packets in its local memory.

Consider the required data rates. Ten bits of data are transmitted for each ASCII character sent between a CPU and the System Monitoring Module: seven bits for the ASCII character, one parity bit, one start bit, and one stop bit. The start and stop bits are used to synchronize the communicating serial ports. If all 30 processors send data at the maximum speed (9600 Baud), a data rate of approximately 28K bytes/second results. Because each character received causes the formation of a two-byte packet, the I/O Processor - System Control Unit parallel port connection must be capable of twice this rate (approximately 56K bytes/seconds). This rate is far below the throughput provided by the parallel

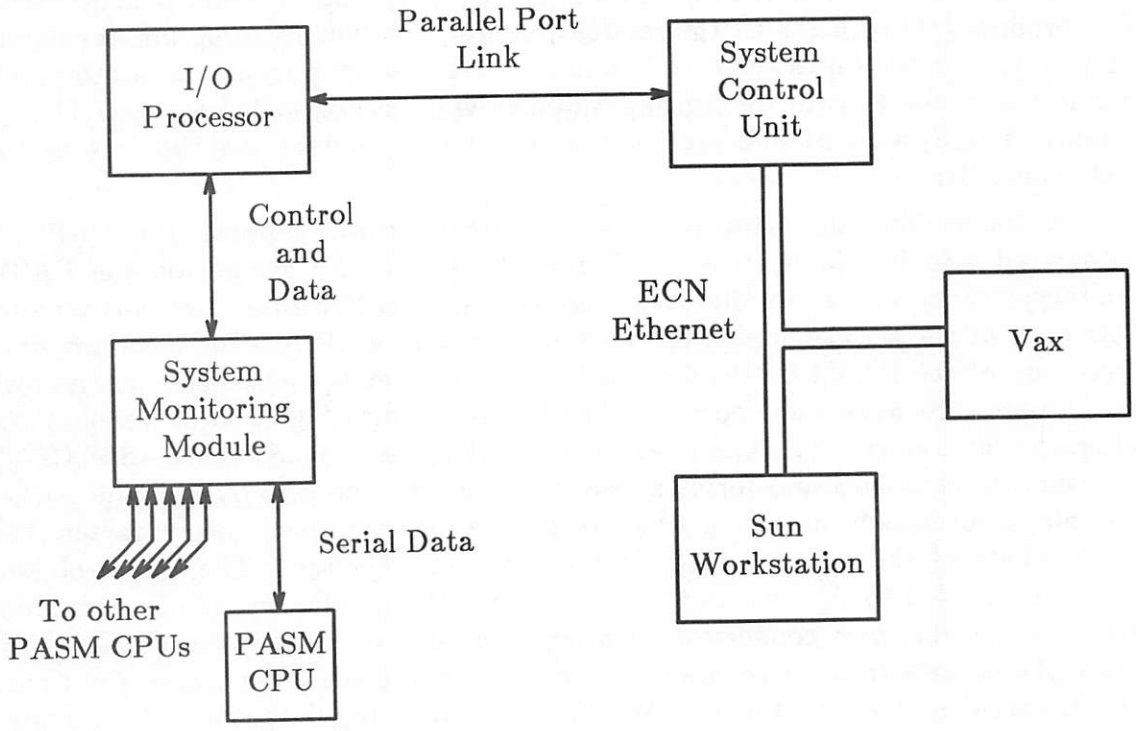


Figure 3. Block diagram of the architectural support for CAPS.

port connections. The Ethernet channel is also capable of this rate.

In addition, the I/O Processor must be capable of these rates as well. When an average single byte memory access is conservatively estimated at one microsecond, 35 memory accesses are permitted per transferred ASCII character. This speed is easily attainable by efficient assembly level programming of the required task. However, the worst case scenario of 30 sending CPUs is very unlikely to be sustained. In addition, the I/O Processor accesses machine instructions two bytes at a time, while the serial ports and parallel port are restricted to single byte accesses. So, 35 accesses per transferred character is a very conservative figure.

While all processors are accessible to the application programmer through CAPS, only the Parallel Computation Unit, MCs, and System Control Unit are of use. Therefore, for applications programmers the worst case scenario mentioned above reduces to 21 sending CPUs. The remaining processors are dedicated to support services for the Parallel Computation Unit, MCs and System Control Unit under operating system control. These service processors are linked to the System Monitoring Module so systems programmers can also take full advantage of CAPS.

Data traveling from the Sun back through the System Monitoring Module originates as keyboard input at a maximum rate of several characters per second. This data rate is negligible compared to data rates to the Sun and was therefore

omitted from the analysis.

When considering the scalability of this system the initial weak point is the I/O Processor - System Monitoring Module pair. The I/O Processor's task of multiplexing data coming from multiple I/O channels must be done in real-time and is limited by the rate it can service the System Monitoring Module's ports. Multiple I/O Processor - System Monitoring Module pairs can be used with larger systems to alleviate this weak point, as the NCube system does by using multiple I/O processing nodes. Again, even if the hardware bottlenecks can be overcome, the user cannot use all the data at once. In general, It is reasonable to accept a maximum number of processors being monitored even if this number is a fraction of the total.

The System Control Unit can also become a bottleneck in an expanded system. During the execution of the parallel program the I/O Processor can be dedicated to supporting monitoring, however, the System Control Unit is running Unix System V and cannot neglect its other duties. It would then become necessary to provide real-time support for the transfer of data from the I/O Processor to the LAN in the form of a dedicated interface to the LAN.

Finally, the user interface could cause a bottleneck. In textural form it is inconceivable for a user to assimilate the data from more than a few processors in real-time and it is a laborious task to go through extensive program traces after execution. The alternative is improved graphical representations of computation and automatic identification of inefficiencies.

Consider the characteristics and/or limitations of an expanded system as described with multiple I/O Processor - System Monitoring Module pairs.

1) I/O would still be possible from all processors because there are multiple I/O Processor - System Monitoring Module pairs.
2) With I/O intensive tasks, it may be possible to visually monitor only a subset of the active processors due to the bandwidth bottleneck at the System Control Unit or LAN.
3) With tasks running on many processors, it may not be possible to convey useful information on all the processors to the user with a single workstation.

Using multiple I/O Processor - System Monitoring Module pairs would permit a large number of processors to be monitored, however, with only a single connection to the LAN and a single interface to the user, only a limited subset of the processors of interest could be actively monitored simultaneously. I/O intensive debugging or debugging where large bursts of information can be generated by the executing program will be a problem for an expanded CAPS. In such cases, a trade-off will exist between the number of processors the programmer wishes to monitor and the detail of the information which the programmer wishes to obtain. For most application types information on the subset of processors of interest will meet the user's needs.

For extremely large numbers of processors (massively parallel) new forms of parallel I/O will have to be created. It will no longer be possible to have any point in the flow of information where there is serialization. In addition, it will

no longer be possible for the programmer to gain but the most general information on the execution in real-time. Most information on the execution will be gained afterwards from the examination of trace files.

Finally, consider the case when multiple users on separate workstations are using the system. If each programmer is monitoring a number of processors that is manageable by his/her own workstation, there is the possibility of performance degradation at the System Control Unit or LAN. The combined number of debug/trace messages from each user's set of monitored processors can saturate the path if the number of users is large enough. With multiple I/O Processor - System Monitoring Module pairs, the System Control Unit becomes the potential bottleneck for I/O traveling from PEs through the System Monitoring Module and I/O Processor to the System Control Unit and to the Sun. To ease this situation, the I/O Processors buffer data when the System Control Unit becomes saturated. As mentioned previously, I/O traveling from the Sun(s) back to the PEs originates as keyboard input at a negligible data rate. Also, program downloading uses different paths within PASM and does not use the path used for interactive monitoring and debugging. The end effect is that programmers experience a higher latency with interactive I/O. The amount of latency caused by a given number of users depend on factors such as I/O required by each user and the software overhead of the I/O Processor and System Monitoring Module. This overhead is difficult to quantify.

7. Next Generation Support

This section describes plans for the next generation of debugging support for PASM. The ultimate goal for this research is the construction of a completely non-intrusive environment with a high-level user interface. The user interface will rely heavily on the graphics capabilities of high resolution workstations to aid the programmer in visualizing the parallel computation. The environment will also automatically identify causes of inefficiencies or contention and point them out to the user. In order for the monitoring of the execution of the program to be non-intrusive the monitoring system must provide substantial hardware support for the identification of events without any modification to the users original source code.

Work toward the development of hardware capable of non-intrusive monitoring is already well underway [MaL89]. The event-action paradigm, a model of the underlying principles of the monitoring process, has been developed. From this model a layered architectural model has been developed and applied to the design of a non-intrusive monitoring system. This sophisticated hardware must be capable of identifying events and tracking the state of the user's program through only a physical connection to the CPU busses of the nodes of the parallel system.

The proposed monitoring system includes a *Central Monitoring Facility* that acts as the user interface (graphic workstation). The Central Monitoring Facility will also be responsible for the coordination and synchronization of the *Special Purpose Hardware Monitoring Units* which are replicated at each node of the parallel system. Additionally, if the network cannot be simulated in software or

if it exhibits non-deterministic behavior, network monitoring hardware will be included. Finally, the components of the monitoring system will be interconnected with a high bandwidth interconnection (e.g. Ethernet) and support hardware for synchronization including a clock line to provide a locally available view of global time.

The Special Purpose Hardware Monitoring Units contain fast comparison logic which compares bus signal patterns with groups of patterns of interest in an event memory. Comparison with these signals allows the identification of program level events such as variables changing value. The Special Purpose Hardware Monitoring units will also analyze predicates involving these program level events such as: "Is the value of the variable zero?" Finally, the Special Purpose Hardware Monitoring units and the Central Monitoring Facility work together to evaluate predicates spanning a number of nodes of the system or concerning the system as a whole such as: "Does variable 'A' equal zero in all nodes." Each Special Purpose Hardware Monitoring unit will include a processor and a high-speed controller to facilitate coordination between Special Purpose Hardware Monitoring units and the identification of predicates.

In order to gain a global ordering of events in the system it is necessary to have a locally available idea of global time. The ability to record the time of occurrence of events (*time-stamp*) is critical to analyzing code execution in parallel machines but is difficult to do with physical clocks [Lam78]. Without time stamps, rebuilding a picture of the execution from the marked events across multiple processors is difficult because, while the events marked on an individual processor are ordered, the events marked across processors are not. The relative ordering of events across processors must be deduced from synchronization points, network accesses, SIMD/MIMD mode switches, etc. To accurately time-stamp events, a global system clock that allows simultaneous access must be present at each processor. Such a global system clock is a necessary but potentially expensive component. Each PASM PE has a 32-bit timer that can be clocked by a single clock line distributed through the machine at a resolution as small as 125 nanoseconds. It is possible to clear and start all timers in SIMD mode so their values proceed identically. Therefore, PASM has a relatively inexpensive global system clock that permits simultaneous access by all PEs.

In addition, work is continuing toward improved user interfaces. Graphical user interfaces show much promise in conveying information on the execution of parallel programs. It may be possible for a programmer to assimilate information about a greater number of nodes with greater detail if the information can be presented in a sophisticated graphical format. The ability to efficiently use parallel machines being designed today depends on the support provided for the development and debugging of applications.

8. Conclusion

This work shows that a small amount of additional hardware can be used to implement a useful remote access and debugging environment. This environment provides remote access to a parallel machine for multiple users and integrates

system features such as downloading code, code development, interactive I/O, and run-time monitoring of programs with sophisticated workstation windowing capabilities.

The implementation of the CAPS environment on the PASM prototype shows that the amount of extra hardware necessary is small and a low degree of system intrusion can be maintained. The added hardware necessary for the CAPS environment implemented on PASM cost on the order of 300 dollars. Of course, the LAN and workstation are not included in this cost.

Work to develop more sophisticated program tracing tools is continuing. These tools will provide more informative graphic displays of program execution to aid in the debugging, testing, and study of parallel algorithms and architectures.

References

[AlB86] R. Allen, D. Baumgartner, K. Kennedy, A. Porterfield, "PTOOL: A semi-automatic parallel programming assistant," *1986 International Conference Parallel Processing,* August 1986, pp. 164-170.

[ArG88] Z. Aral, I. Gertner, "Parasight: A high-level debugger/profiler architecture for shared-memory multiprocessors," *ACM 1988 International Conference on Supercomputing,* pp. 131-139, July 1989.

[Bat88] P. Bates, "Distributed debugging tools for heterogeneous distributed systems," *8th International Conference on Distributed Computing Systems,* June 1988, pp. 308-316.

[BaW83] P. Bates, J. Wileden, "High level debugging of distributed systems: The behavioral abstraction," *Journal of Systems and Software,* Vol. 3, No. 4, April 1983, pp. 255-264.

[BrC89a] E. C. Bronson, T. L. Casavant, L. H. Jamieson, "Experimental analysis of multi-mode fast fourier transforms on the PASM parallel processing system," *1989 IEEE International Conference on Acoustics, Speech, and Signal Processing,* May 1989, pp. 2540-2543.

[BrC89b] E. C. Bronson, T. L. Casavant, L. H. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *1989 International Conference on Parallel Processing,* August 1989.

[BrM89] W. Brantley, K. McAuliffe, and T. Ngo, "RP3 performance monitoring hardware," in *Instrumentation for Future Parallel Systems,* edited by Margeret Simmons, Addison Wesley, to appear 1989.

[Cha86] R. N. Charette, *Software Engineering Environments,* McGraw Hill, 1986.

[Che84] T. E. Cheatham, "Program reusability through program transformation," *IEEE Transactions on Software Engineering,* Vol. 10, No. 9, September 1984, pp. 620-627.

[ChZ85] D. R. Cheriton, W. Zwaenopol, "The distributed V kernel and its performance for diskless workstations," *9th Symposium on Operating Systems,* October 1985, pp. 128-139.

[CoA86] R. P. Cook, R. J. Auletta, "Starlite, a visual simulation package for software prototyping," *ACM SIGPLAN 86 Symposium on Language Issues and Programming Environments,* Proceedings appear as SIGPLAN Notices special issue Vol. 21, No. 7, 1986, pp. 102-110.

[Cou87] A. L. Couch, "Seecube User's Manual," *Technical Report, CS Department, Tufts University*, 1987

[DeS86] N. Delisle, M. Schwartz, "A programming environment for CSP," *ACM SIGPLAN 86 Symposium on Language Issues and Programming Environments,* Proceedings appear as SIGPLAN Notices special issue Vol. 21, No. 7, 1986, pp. 34-41.

[Enc86] Encore Computer Corporation, "ANNEX hardware installation guide," and "ANNEX User's Guide," *Document #'s 716-02887 and 716-02886,* 1986.

[FiC88a] S. A. Fineberg, T. L. Casavant, T. Schwederski, H. J. Siegel, "Nondeterministic instruction time experiments on the PASM system prototype," *1988 International Conference on Parallel Processing,* August 1988, pp. 444-451.

[FiC88b] S. A. Fineberg, T. L. Casavant, H. J. Siegel, "Experimental evaluation of SIMD PE-mask generation and hybrid mode parallel computing on multiprocessor systems," *Technical Report TR-EE 88-55,* School of Electrical Engineering, Purdue University, revised July 1989.

[Fle85] Flexible Computer Corporation, "The Flex/32 MultiComputer: System overview," *Report No. 030-0000-002,* Flexible Computer Corporation, 1985.

[GoM82] G. H. Goble and M. H. Marsh, "A dual processor VAX 11/780," *9th Annual Symposium on Computer Architecture,* April 1982, pp. 291-298.

[HaM86] J. P. Hayes, T. N. Mudge, Q. F. Stout, and S. Colley, "Architecture of a hypercube supercomputer," *1986 International Conference on Parallel Processing,* August 1986, pp. 653-660.

[Lam78] L. Lamport, "Time, clocks and ordering of events in a distributed system," *Communications of the ACM,* Vol. 21, No. 7, July 1978, pp. 558-565.

[LeM87] T. J. LeBlanc, J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers,* Vol. C-36, No. 4, April 1987, pp. 471-482.

[MaL89] D. C. Marinescu, J. E. Lumpp, Jr, T. L. Casavant, H. J. Siegel, "A model for monitoring and debugging parallel parallel and distributed software," *13th Annual International Computer Software and Applications Conference (COMPSAC89),* September 1989.

[MaW86] K. Marzullo, D. Wiebe, "Jasmine: A software system modeling facility," *ACM SIGPLAN 86 Symposium on Language Issues and Programming Environments,* Proceedings appear as SIGPLAN Notices special issue Vol. 21, No. 7, 1986, pp. 121-131.

[McD75] G. McDaniel, "Metric: A kernel instrumentation system for distributed environments," *6th Symposium on Operating Systems Principles,* November 1975, pp. 93-99.

[MiC88] B. P. Miller, J. D. Choi, "Breakpoints and halting in distributed programs," *8th International Conference on Distributed Computing Systems,* June 1988, pp. 316-323.

[MiY87] B. P. Miller, C-Q Yang, "IPS: An interactive and automatic performance measurement tool for parallel and distributed programs," *7th International Conference on Distributed Computing Syatems,* September 1987, pp. 482-489.

[Mot83] Motorola, "VMEbug: Debugging package user's manual," *MVMEbug/D3,* Motorola, Inc., 1983.

[PoM83] M. L. Powell, B. P. Miller, "Process migration in DEMOS/MP," *9th Symposium on Operating Systems,* October 1983, pp. 110-119.

[Pra85] T. W. Pratt, "Pisces: An environment for parallel scientific computation," *Software,* Vol. 2, No. 4, July 1985, pp. 7-20.

[ScG86] R. W. Scheifler and J. Gettys, "The X window system," *ACM Transactions on Graphics,* Vol. 5, No. 2, April 1986, pp. 79-109.

[Sch81] R. D. Schuffenbauer, "Interactive debugging in a distributed computing environment," *Technical Report MIT/LCS/TR 264*, MIT, September 1981.

[ScN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "Design and implementation of the PASM prototype control hierarchy," *Second International Supercomputing Conference,* May 1987, pp. 418-427.

[SeR85] Z. Segall, L. Rudolph, "PIE: A programming and instrumentation environment for parallel processing," *Software,* Vol. 2, No. 6, November 1985, pp. 22-37.

[Sie85] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath, Lexington, MA, 1985.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, N. J. Davis IV, "An overview of the PASM parallel processing system," in *Tutorial: Computer Architecture,* edited by D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and P. B. Furht, IEEE Computer Society Press, 1987, pp. 387-407.

[SmK85] D. Smith, G. Kotik, S. Westfold, "Research on knowledge-based software engineering environments at Kestrel Institute," *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, November 1985, pp. 1278-1295.

[SnS86] L. Snyder, D. Socha, "Poker on the COSMIC Cube: The first retargetable parallel programming language and environment," *1986 International Conference Parallel Processing,* August 1986, pp. 628-635.

[Sun86] Sun Microsystems, "Sun system overview," *Sun Microsystems Manual,* Sun Microsystems, Inc., 1986.

[Swe85] R. E. Sweet, "The Mesa programming environment," *ACM SIGPLAN 85 Symposium on Language Issues and Programming Environments,* Proceedings appear as SIGPLAN Notices special issue Vol. 20, No. 7, 1985, pp. 216-229.

[Tay83] G. S. Taylor, "Arithmetic on the ELXSI system 6400," *6th Symposium on Computer Arithmetic,* May 1983, pp. 110-115.

[Tei84] W. Teitelman, "A tour through Cedar," *Software*, Vol. 1, No. 2, April 1984, pp. 44-73.

[ThG86] B. Thomas, R. Gurwitz, J. Goodhue, D. Allen, and M. Beeler, "Butterfly parallel processor overview," *BBN report no. 6148,* BBN Advanced Computers, Inc., 1986.

[Van87] A. van Tilborg, ed., *Workshop on instrumentation for distributed computing systems*, IEEE Computer Society and ACM, January 1987.

[Wat85] R. C. Waters, "The programmer's apprentice: A session with KBEmacs," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 11, November 1985, pp. 1296-1317.

THE IMPLEMENTATION OF AIDE: A SUPPORT ENVIRONMENT FOR DISTRIBUTED OBJECT-ORIENTED SYSTEMS

Rodger Lea[1] and Jonathan Walpole[2]

Computing Department,
Lancaster University,
Bailrigg,
Lancaster,
LA1 4YR,
U.K.

ABSTRACT

The basic properties of encapsulation and message passing make the object oriented paradigm inherently suitable for programming distributed systems. However, despite these properties, projects which have addressed the problem of building distributed object oriented systems have often been forced to rely on some significant degree of homogeneity, either at the hardware, system software or object model level.

In this paper we describe the implementation of the Aide system which provides a *generic* set of mechanisms for object communication, grouping and flexible interaction both locally and across machine boundaries. A salient feature of the Aide system is that it is implemented, separately from the host operating system kernel, on a co-processor which acts as an intelligent front end to the host machine. Hence, Aide can support a number of different object models and can co-exist with a variety of host operating systems. An important aspect of the Aide project has been the experience it has given us with the use of hardware support for object interaction in distributed systems.

1. Introduction

In recent years the object oriented paradigm has become widely accepted as a model for dealing with the complexities of many aspects of computing. Its benefits as a programming language model, as a system and database structuring tool and as a vehicle for software reuse are now well recognised. In the area of distributed computing the object oriented paradigm is particularly useful because its explicit concepts of encapsulated objects and message passing mean that programming distributed applications is conceptually no more complex than programming centralised applications.

As a result of this apparent suitability for distribution, several projects have addressed the problem of distributing object oriented systems [Black et al. 1987, Schelvis et al. 1988,

---

1 Author's current address is: Chorus Systemes, 6 Avenue Gustave Eiffel, F-78183, Saint-Quentin en Yvelines, Cedex, France.

2 Author's current address is: Oregon Graduate Center,19600 N.W. Von Neumann Drive, Beaverton, OR 97006-1999, USA.

Decouchant et al. 1988, Liskov 1988, Bennett 1987, McCullough 1987]. However, in many cases this task has turned out to be non trivial. It is significant that each system has had to reimplement mechanisms to support object communication, location, migration, replication and grouping. We argue that this is a result of implementing such mechanisms at too high a level, and that they should be provided, not at the language or application level, but by the underlying system.

In this paper we describe the implementation of the Aide system [Lea 1989] which provides a generic set of mechanisms for object communication, grouping and flexible interaction both locally and across machine boundaries. In contrast to a number of current object oriented operating system projects [Hermann et al. 1988, Bernabeu et al. 1988, Nicol et al. 1989, Jones et al. 1986]. Aide provides such support, separately from the host operating system kernel, on a co-processor which acts as an intelligent front end to the host machine. This approach allows Aide to support a number of different object models and allows it to co-exist with a variety of host operating systems. The experience gained in designing and implementing Aide has given us valuable insight into the advantages and disadvantages of using hardware support for object interaction in distributed systems.

The paper consists of the following sections. Section 2 outlines the requirements which must be met by a distributed object oriented support environment. The basic facilities offered by the Aide system are presented in section 3, and the Aide implementation environment is discussed in section 4. Section 5 presents our experiences with the Aide system and discusses both the advantages and disadvantages of our approach. Finally, section 6 concludes the paper.

2. Requirements for Distributed Object Oriented Systems

Despite the fact that the object oriented programming paradigm is not complicated by distribution at a conceptual level, there are a number of practical issues which must be addressed in order to support object interaction in a distributed environment. These include the following:

Object Location:

> The support system must provide facilities for locating objects in a distributed environment. This problem is complicated if the dynamic migration of objects is supported.

Distribution Transparency:

> Appropriate levels of distribution transparency must be identified and implemented, especially in terms of object location.

Replication:

> To support highly available applications, the support system should provide facilities for replicating objects on separate nodes.

Communication:

> The support system must provide efficient facilities for inter-object communication both locally and across machine boundaries.

Grouping:

The use of implicit or explicit class/type hierarchies in object oriented systems results in well defined patterns for object interaction. These usually take the form of either, interaction between instances or interaction between an instance and its corresponding class or 'concrete' data type. This latter form of interaction can also involve interaction between classes and their super-classes (recursively). Since several object oriented models (e.g. [Goldberg et al. 1983]) take the view that classes are also objects, the interaction between instances, classes and super-classes can be treated uniformly. In this case it is necessary to identify these interaction groupings and where possible to make intra-group communication as efficient as possible.

In addition to meeting the requirements listed above, a decision must also be made concerning the location of such support mechanisms. This decision has important repercussions, especially in relation to the efficiency of the system and the level of heterogeneity which can be supported. If the support mechanisms are built into the operating system kernel [Decouchant et al. 1988, Jones et al. 1986] or are provided by a specialised virtual machine [Goldberg et al. 1983] considerable porting or reimplementation effort will be required for heterogeneous environments. In this paper we suggest an alternative approach in which the support mechanisms are implemented in a separate 'add-on' module which operates in parallel with the host operating system. This module provides hardware support for object interaction and consequently, it enhances the efficiency of the system. As long as this module does not rely too heavily on the host system it will also be possible to support efficient portability between heterogeneous nodes.

3. The Aide Support Mechanisms

In order to meet the requirements listed above the Aide system provides a number of basic mechanisms. Access to these mechanisms is provided via a well defined Aide system call interface which allows the host system to make use of the services provided by Aide. At the lowest level of the Aide system, a secure, lightweight protocol is implemented. This guarantees host to host delivery without incurring the overheads of more sophisticated protocols such as TCP/IP. Above this layer, an object communication facility is provided which supports both local and remote object messaging and invocation, and which uses a location transparent naming mechanism. The particular functions offered by the Aide system include: object registration, naming, messaging, relocation, grouping, replication and monitoring. These functions are addressed in the following subsections.

3.1 Initialising the Aide System

During the initialisation phase of each Aide system a number of messages are exchanged with other Aide systems on the network to determine the overall system configuration. Once the Aide system has been incorporated into the network in this way it becomes available to service requests. These requests can come either from local host objects or from remote objects.

3.2 Registration

Object registration is supported using the Aide system call:

my_id = register (Obj_name, info_block)

which takes a user specified name and a pointer to an information block as parameters. The information block describes the object to be registered, and contains information including the object's name, type, size, signature, attributes and mobility. Upon successful registration, the call returns a system wide unique object identifier. The object can then access (and be accessed by) other objects both locally and remotely.

3.3 Naming

When an object registers, it is assigned a unique internal identifier which consists of a local port number concatenated with the local node address. Aide maintains a number of data structures for each object under its control, one of which contains the mapping from the object's user specified name to its unique identifier.

3.4 Messaging

Message passing is supported by two Aide system calls. The first:

ack = message (Obj_name, my_id, message, block_flag, msg_type)

is used to send a message to a designated object. The returned value can acknowledge one of three possible events: that the message has been queued for attention by the local Aide system; that the recipient object has been located; that the message has arrived on the node of the recipient object. Note that no attempt is made to acknowledge the receipt of the message by the recipient object. This task is left to higher level protocols of which Aide has no direct knowledge.

The second message system call:

result = receive (my_id; place_here, max_amount, type)

is used to receive messages. The effect of this call is the to copy the data of the message into the designated 'place_here' area of memory. The type field designates the type of messages which will be accepted by the receiver. This value may be defined either by Aide or by higher level software.

The protocol used for object location is optimised, and uses a number of hint tables to try to locate an object quickly. First, Aide searches a table of system-wide, well known objects which have identified themselves as being special during registration (this causes their location to be recorded at every node). If this initial search is unsuccessful then Aide searches the cache table associated with the sending object. This maintains a list of the locations of the most frequently accessed local and remote objects. If this is also unsuccessful then Aide searches a list of local objects to determine whether the required object is local. Finally, if all these approaches are unsuccessful, a broadcast mechanism is used to determine the object's location. Forwarding lists are also maintained to keep track of objects which have migrated.

3.5 Relocation

Aide provides support for moving objects from one node to another via the system call:

result = move (my_id, to, from).

This causes the designated object to be reregistered at the specified Aide site. However, this call does not cause the actual object to be removed from the local site since this is dependent on higher level protocols.

3.6 Monitoring

Each registered object has an associated monitor block on the network access unit (see section 4). This allows Aide to maintain information concerning an object's registration time, number of messages in, number of messages out, most frequently messaged objects and their location, average message size and average interval between messages. This information is used by Aide in order to make a number of optimisations, and can also be accessed by higher layers. An example of an optimisation performed by Aide is the caching of the most frequently messaged objects and their locations. This is used to speed up messaging by circumventing the requirement to locate each object as it is referenced.

3.7 Grouping

The previous sections have described facilities which support object interaction purely at the communication level. However, to provide adequate support for a variety of object oriented models, it is also necessary to augment these basic facilities with facilities to support groupings and hierarchies of objects. These groupings may be either implicit or explicit.

Implicit groupings can arise because of the tendency for objects to cluster to form short term execution environments during the execution of a task. Once a particular task has completed, the initiation of a new task reconfigures the working set of objects.At any point, this working set can be determined by Aide using the monitoring information stored with objects. Such information can be used by Aide to optimise the performance of the system by initiating the migration of frequently accessed remote objects. It is important to note that such groupings may arise, either as a result of interactions between a number of instances, or as a result of the interactions between classes in a class hierarchy. At the Aide level both types of interaction are equivalent and are optimised using the same techniques.

Aide also supports facilities for grouping objects explicitly. The Aide system call:

group_id = create_grp (grp_name, info_block)

can be used to create a named group. Two functions: join_group and leave_group are provided to add/delete objects to/from a group.

A further explicit group facility is provided to support distributed replication of objects which can enhance both reliability and performance. This facility uses the system call:

result = replicate (my_id, copies, locations)

which, makes use of the move system call to create copies of objects at different nodes in the system.

The protocols for object location are also augmented to allow group communication, and facilities are provided to allow objects to be located based on information relating to their type, signature, or attributes. These mechanisms provide support for a variety of flexible binding policies. On registration, each object provides extensive type information to Aide. This information includes the object's type, where it is derived from, who it uses to locate methods, and what interface it exports to the system. Thus, when an object invokes a method, either the method is defined locally, in which case it is executed immediately, or a search must be performed to locate the method. This search is based on various information which Aide associates with the object, and makes use of Aide's object location facilities. As well as maintaining an object's registered information, Aide also accumulates information relating to the location of methods as they are invoked. In this way, Aide becomes able to circumvent the process of location chaining, and messages the correct objects directly. For further details of the Aide implementation see [Lea 1989].

4. The Aide Implementation Environment

Aide operates as a co-module to the host system and is designed to work on a separate co-processor which has direct control of the network. This co-processor is commonly referred to as a network access unit (NAU). Aide interfaces to the host machine using an area of dual ported RAM which is resident on the NAU. A simple interface protocol is implemented to handle contention for access to this area from both the host and the NAU. This is achieved by making the host pass pointers to data which needs to be moved onto the NAU and then by allowing Aide to use these pointers to read from host memory when that data is required. This relies on some area of the host memory being designated as system or bus accessible. If this is not possible then Aide can be configured at boot time to assign an area of the dual ported RAM to message data.

Aide is currently implemented on a CMC ENP-10 intelligent network access board [CMC 1986]. This comprises a Motorola 68010 processor, a Lance ethernet chipset [AMD 1982] and 512k of RAM, of which 128k is configured as dual-ported and is available to the system bus. The ENP-10 board is supplied with the K1 kernel [CMC 1986] which provides 4 basic routines: configuration, send and receive of ethernet packets, and basic ethernet monitoring statistics.

The current development machine, to which the board is hosted, is a single card M68010 processor with memory management facilities. These are provided by a Microsys CPU-07 [MicroSys 1985] board which runs OS-9 [MicroSys 1985]. The backplane bus is VME [Motorola 1982].

The remaining nodes in the test environment are Sun 3 workstations running Unix. Since the Sun workstations do not contain an intelligent network access unit comparable to the ENP-10, each workstation runs a suite of Aide simulation software. This software uses the Network Interface Tap protocol [Sun 1985] which is layered on top of the Unix raw socket protocol [Sun 1985]. This interface was chosen because it allows Unix processes to read and write ethernet packets. Sockets are used to allow user objects to access the Aide simulation.

Both the ENP-10 and Sun implementations provide protocol support for the ANSI 802.3 level 2 layer [ANSI 1985]. This effectively means that messaging is at the basic ethernet packet layer. It was decided early in the project that more efficient communications would be realised if the message subsystem aimed to provide a reasonably secure lightweight protocol and, whilst guaranteeing host delivery, made no attempt to offer more sophisticated protocols. Although this effectively created a closed system it was felt that the

benefits from low protocol overheads would more than offset the cost of using protocol converters within gateways in order to access other networks. The supporting rationale is that the majority of object interaction is performed within a geographically localised area.

5. Experiences with Aide

Our experiences with the Aide project fall into two general categories. The first concerns the use of the object oriented paradigm for distributed systems design and implementation. The second concerns the architectural approach taken by Aide.

Our experience with the object oriented paradigm has been positive. In particular, the object interface and typing information supported by the object oriented paradigm can provide valuable hints to the support system. In Aide, some of this higher level information has been pulled down onto the front end board in order to enhance the system's flexibility. Whilst this approach has not been entirely straight forward (see below), we consider it to be vital for supporting flexible and efficient object interaction in a distributed environment. This is an area which we hope to study further in the future.

The architectural approach taken by the Aide project has resulted in a number of advantages. Firstly, the use of the co-processor architecture has the distinct performance advantage that once a message has been registered with the communications board tasks such as protocol conversion, object location and messaging can all take place without placing any load on the host processor. At present we have not quantified this benefit. However, it is hoped that an implementation of a complete Aide system (i.e. with several other nodes running Aide co-processors) will provide a framework within which to judge these benefits.

Secondly, the ability to off-load distribution mechanisms to a separate co-processor allows a high degree of transparency to be provided to the host system in terms of messaging, replication, fault tolerance and heterogeneity.

Thirdly, the use of a co-processor allows complex mechanisms for object naming, location and invocation to be supported, and finally, the architecture allows heterogeneity to be supported at the hardware and system software levels by making a heterogeneous host look like a homogeneous Aide node to the rest of the system. In effect, the co-processor acts as a buffer to the rest of the distributed system, hiding individual node heterogeneity.

However, in the current implementation the isolation of the Aide system from its host also has some disadvantages. The storage of object type and attribute information within the Aide system restricts the operation of the host system. In particular, constraints on the flexibility of the attribute mechanism mean that it is slow and difficult for the host to change the attributes associated with objects. We are hoping to experiment with policies for caching object type and attribute information within Aide in order to relax these restrictions.

A second disadvantage arises because of the isolation of the communication mechanisms, and in particular the location protocols, from the host. This makes it is difficult for the host system to play any role in these activities and makes it especially difficult to support an integrated language/system model such as that presented by the Emerald project [Black et al. 1986].

Furthermore, the current implementation forces local communication to use the same mechanisms as remote communication. As a result, even with hardware support, local communication is no faster than many message passing systems. Whilst this could be

improved, such improvements appear to compromise the original goals of a uniform communication model (both local and remote) and support for heterogeneity.

The current implementation of Aide is still a research testbed and, as such, is both inefficient and subject to change. Nevertheless, the Aide communication primitives are comparable to those of a number of other message passing systems: a local call with acknowledgement of 'queued for attention' executes in 3 ms; a local send and receive with 100 bytes of data takes 8 ms; a remote send and receive executes in 12 ms (although this is subject to network load).

The performance of the object location primitives varies according to the specific location protocol used, the network load and the remote node load. However, round trip delays between 18 and 120 ms have been recorded. A large proportion of this delay is caused by the use of two stages of UDP sockets on the nodes which use the Sun simulation.

It should be stressed however that the current implementation is not optimised and that a number of inefficiencies are due to the development environment rather than the underlying communication and location mechanisms. More specifically:

(i) Communication between the host and the Aide system is currently implemented using a mailbox paradigm which supports a high degree of autonomy between Aide and the host, but which slows down communication.

(ii) The location protocol is not optimal. The use of hint lists is at present simplistic and provides little help in locating the majority of objects. Nevertheless, the Aide performance figures compare favourably with those reported for the Argus system [Liskov et al. 1987] and are between 2 and 8 times slower than those reported for Amoeba [Van Renesse et al. 1988].

5.1 The Use of Aide in the Oscar Project

The Aide system has been used as a basis for the implementation of a distributed Object System for Control Applications and Robotics (OSCAR) [Shepherd et al. 1988a]. The Oscar system is designed to overcome the problems of traditional centralised approaches to implementing control and robotic applications in factories by using a model based on the concept of active objects (each active object is a self contained process). Oscar uses a hybrid model based on aspects of both inheritance and prototypes. This provides an ideal testbed for Aide since the model requires both static and dynamic location of methods.

Aide performs several roles in the Oscar system. At the system definition phase Aide is used to gather information on the configuration and characteristics of the available network. This information is used to guide the development engineer when developing objects and interaction patterns. Once the system is defined Aide is used to distributed the individual objects to the required host nodes and to initiate their execution. This also requires the use of a special Oscar control object which resides at each node and is used to receive the migrating objects and to install them at that node. The objects are then initiated which involves registration with Aide followed by the execution of their main algorithm. Once this phase has been completed, Aide is used to support the run time interaction between the active objects which are now distributed over the various host machines (using the techniques discussed in previous sections).

The use of Aide as a basis for the Oscar project simplified the design and implementation of Oscar considerably, and demonstrated the power of the Aide approach. For further details of the Oscar project see [Shepherd et al. 1988a, Shepherd et al. 1988b].

6. Conclusion

In this paper we have described the Aide system which has been designed and implemented at the University of Lancaster over the past three years. Aide provides a generic set of mechanisms for object communication, grouping and flexible interaction in a distributed environment. However, unlike current distributed object oriented operating system projects, the Aide system is implemented separately from the host operating system on a co-processor which acts as a front end to the host node. This co-processor has full control over network access and shares an area of dual ported RAM with the host node.

The experience gained in implementing the Aide system has highlighted two important lessons. Firstly, the use of hardware support in the form of a powerful front end processor provides a key to supporting efficient and flexible object interaction in distributed systems. Secondly, we have found the object oriented paradigm to be useful in designing and building distributed systems. In particular, the object interface and typing information supported by the model can provide valuable hints to the support system. In Aide, some of this information has been pulled down onto the front end board in order to enhance the system's flexibility. In our future research we hope to investigate further the level to which object model information can be pushed down into distributed systems architectures.

Acknowledgements

This work was supported by the U.K. Science and Engineering Research Council.

References

AMD 1982
AMD, "The AM7990 Family Ethernet Node," Advanced Micro Devices, November 1982.

ANSI 1985
ANSI, "ANSI/IEEE Standard 802.3 1985 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification," 1985.

Bennet 1987
Bennett, J. K, "The Design and Implementation of Distributed Smalltalk," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Special Issue of SIGPLAN Notices*, vol 22, no. 12, pp. 318-330,.Orlando, FL, October 4-8, 1987.

Bernabeu et al. 1988
Bernabeu, J., Khalidi, Y. A, Ahamad, M., Appelbe, W. F., Dasgupta, P., LeBlanc, R. J. and Ramachandran, U., "Clouds - A Distributed Object Oriented Operating System: Architecture and Kernel Implementation," *Proceedings of the 88'EUUG Conference*, Casais, Portugal, October 1988.

Black et al. 1986
Black, A., Hutchinson, N., Jul, E., and Levy, H., "Object Structure in the Emerald System," *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '86), Special Issue of SIGPLAN Notices*, vol. 21, no. 11, Portland, Oregon, September 29-October 2, 1986.

Black et al. 1987
Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, January 1987.

CMC 1986
CMC, "ENP-10 Ethernet Node Processor," CMC Technical Report, Communication Machinery Corporation, Santa Barbara, USA, 1986.

Decouchant et al. 1988
Decouchant, D., Duda, A., Freyssinet, A., Riveill, M., Rousset de Pina, X., Vandome, G., and Scioville, R., "Guide: An Implementation of the COMANDOS Object Oriented Distributed System Architecture on Unix," *Proceedings of the 88'EUUG Conference*, Casais, Portugal, October 1988.

Goldberg et al. 1983
Goldberg, A., and Robson, D., "Smalltalk-80: The Language and its Implementation," Addison-Wesley, 1983.

Hermann et al. 1988
Hermann, F., Armand, F., Rozier, M., Gien, M., Abrossimov, V., Boule, I., Guillemont, M., Langlois, S., and Neuhauser, W., "CHORUS: A New Technology for Building UNIX Systems," *Proceedings of the 88'EUUG Conference*, Casais, Portugal, October 1988.

Jones et al. 1986
Jones, M. B. and Rashid, R. F., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *Proceedings of the Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '86), Special Issue of SIGPLAN Notices,* vol. 21, no.11, Portland, Oregon, September 29-October 2, 1986.

Lea 1989
Lea, R., "Network Support for Distributed Objects: Coping with Heterogeneity in Models and Architectures," Ph.D. Thesis, University of Lancaster, March 1989.

Liskov et al. 1987
Liskov, B., Curtis, D., Johnson, P., and Scheifler, R., "Implementation of Argus," *Proceedings 11th ACM-SIGOPS Symposium on Operating Systems Principles,* Austin, Texas, pp.111-122, November 1987.

Liskov 1988
Liskov, B., "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300-312, March 1988.

McCullough 1987
McCullough, P. L., "Transparent Forwarding: First Steps," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Special Issue of SIGPLAN Notices,* vol. 22, no. 12, pp. 331-341, Orlando, FL, October 4-8, 1987.

MicroSys 1985
MicroSys, "MicroSys CPU-07 User's Manual," MicroSys Electronics GmbH, Munich, West Germany, 1985.

Motorola 1982
Motorola, "VME Bus Specification Manual MVMEBS/D1," Motorola Semiconductors Inc., August 1982.

Nicol et al. 1989
Nicol, J. R., Blair, G. S., Walpole, J., and Malik, J., "COSMOS: an Architecture for a Distributed Programming Environment," *Computer Communications*, June 1989.

Van Renesse et al. 1988
Van Renesse, R., Mullender, S. J., and Tanenbaum, A. S., "Performance of the World's Fastest Distributed Operating System," *ACM SIGOPS Review,* vol. 22, no. 4, October 1988.

Schelvis et al. 1988
Schelvis, M., and Bledoeg, E., "An Implementation of Distributed Smalltalk," *Proceedings of the European Conference on Object Oriented Programming,* Oslo, Norway, June 1988.

Shepherd et al. 1988a
Shepherd, D., Coote, S., Gallagher, J., Lea, R., Mariani, J., and Scott, A., "An Object Oriented Approach to the High Level Programming of Distributed Process Control Applications," *Proceedings of the 3rd International Conference on Computer-Aided Production Engineering*, Michigan, U.S.A., June 1988.

Shepherd et al. 1988b
Shepherd, D., Coote, S., Gallagher, J., Lea, R., Mariani, J., and Scott, A., "The Design and Implementation of a Distributed Control and Robotics Application based on a Model of Active Objects," *Proceedings of the U.K. IT 88 Conference,* Swansea, U.K., July 1988.

Sun 1985
Sun, "Sun Unix Manual, Release 3," Sun Microsystems Inc., October 1985.

Experience with Implementing and using an Object Oriented Distributed System

Dominique DeCouchant, Michel Riveill, Sacha Krakowiak*
LGI-IMAG, BP 53x 38041 Grenoble cedex, France

Chris Horn, Edward Finn, Neville Harris †
Dept. of Computer Science, Trinity College Dublin, Ireland

Abstract

This paper describes early experience in the implementation and usage of an object-oriented, distributed operating system. This experience has been gathered on two different implementations of the same system architecture. Section 1 is a general description of the architecture. Section 2 describes an implementation on a bare machine. Section 3 describes an implementation of top of Unix [1]. Section 4 is a summary of the experience gathered in building and using the system.

1 The Comandos Architecture

Comandos is a distributed object-oriented platform for the *Construction and Management of Distributed Open Systems*. The platform is being designed and implemented on several host systems jointly by Trinity College Dublin, the University of Grenoble, Bull, INESC/University of Lisbon, Siemens, Nixdorf, Chorus Systèmes and other academic and industrial partners, and is partially funded under the ESPRIT programme. Its goal is to design and construct an integrated platform for programming distributed applications which may manipulate persistent data [Horn87] [Horn89]. In particular, the project investigates a distributed operating system architecture, in which many of the problems of distributed resource allocation are hidden from the user. The main guidelines of the project are as follows:

Language support An operating system may be viewed as an execution environment for programming languages. This environment may be provided to the user as a set of primitives, or as a full-fledged language whose run-time environment is supported by the system. We have adopted both approaches. The first provides language independence, and the primitives may be supplied as library calls. The second approach provides a better integration of the system and applications, and we have designed a strongly typed, object-oriented programming language for the expression of distributed applications.

Object storage The system includes a permanent repository for objects, which may be viewed as a substitute for the traditional file system. Objects thus provide a unifying view, since they may be regarded both as a support for procedural and data abstraction and as long-term storage units: this is the "persistent programming" approach. Objects may be composed to form arbitrarily complex structures.

*e-mail: {decoucha,riveill,krakowiak}@imag.imag.fr
†e-mail: {horn,finn,harris}@cs.tcd.ie
[1] trademark of ATT

Distributed virtual machine The main computational abstraction provided to the users is the job, a multiprocessor virtual machine in which an arbitrary number of concurrent activities operate on a collection of objects. Since the objects may be located on different nodes, a job may be distributed; however, this distribution is usually hidden from the user. A user may define several jobs, which communicate through shared objects.

A job provides an arbitrary number of sequential threads of control, called activities. It also provides an addressing window on the global object space, by mapping a set of objects which are shared by the activities. The set of objects mapped within a job is called its context. The composition of the context may dynamically change as objects are mapped or unmapped. Communication between activities within the same job or in different jobs takes place through shared objects. Synchronization constraints can be attached to a shared object in order to control the interactions of concurrent activities.

A job may span several physical nodes; actually, it may dynamically extend itself or shrink, according to the pattern of object invocations. Distribution is usually hidden from the user of a job.

The addressing and execution space provided by jobs may be viewed as a "cache" for a permanent repository where objects are stored, but cannot be directly addressed. This concept is related to file mapping and has been introduced e.g. in the Apollo/Domain [Leach83] system.

The object memory is therefore implemented as a two-level store. At the lower level, a Storage Subsystem *SS* is in charge of the long-term storage of persistent objects. At the upper level, a Virtual Object Memory *VOM* supports the execution of jobs (i.e. objects bound to jobs for execution are addressed in the VOM). Both VOM and SS are distributed. Note that an object mapped in VOM may be actually loaded and executed on any physical node, as long as its executable code is compatible with the processor of that node.

An object is named by a reference, i.e. a system-wide unique identifier from which the location of the object can be determined. A reference contains object identity information and a location hint.

Shared objects introduce the need of a synchronization mechanism. In accordance with the object model, we choose to associate this mechanism to objects, not as separate synchronization primitives within activities [Decouchant88b]. Thus an object is entirely self-contained, including the specification of synchronization.

2 Implementation on a bare machine

The main reasons for developing Comandos on a bare machine were to gain experience in the development of a distributed operating system, and also to develop the Comandos kernel without constraints of an underlying system with a view to functionality and overall performance [Marques88].

The prototype developed at Trinity College Dublin is called the *Oisin* kernel and was initially implemented on a network of NS32000 based machines (known locally as Trinity Workstations: these provide general virtual memory support, ethernet, SCSI disks and a number of serial communications channels). It is since being ported to Digital μVax-IIs.

The kernel is partitioned into a number of components. As mentioned above, there is the VOM component - responsible for object location and virtual memory - and the SS - responsible mainly for persistent storage of objects. Other components are the *AM* - responsible for job and activity objects in the system i.e. the active components of the Comandos system; the *CS* - responsible for communications; and the i/o subsystem - responsible for devices and i/o operations.

This division of functionality among the components of the kernel has led to much modularity in implementation and a considerable amount of parallel development and testing of components of the kernel.

The runtime environment in user mode of the Comandos system has been developed in conjunc-

tion with the kernel. The interfaces between the runtime and the kernel have been defined and the object support and invocation has been fully working since a very early stage in the development.

A stable version of the Oisin kernel has been in operation for a number of months on the TWS. We are also porting this version of the Comandos kernel to a μVax. The experience gained in the initial prototype meant that this task has proved less difficult than expected. Again the modularity of the kernel proved useful in that each component could be ported individually and tested for the most part.

Although defined in an object oriented fashion and accessed from user applications as objects, it was not developed using an object oriented language. The development of the kernel has been in Modula2 with exported interfaces between the components of the kernel.

3 Implementation on Unix

Guide is an implementation of the Comandos architecture based on Unix, developed jointly by Bull and LGI at Grenoble. We decided to produce a Unix-based implementation in order to have a working prototype within a short time, ready for experiments on the object model and computational scheme, at the price of some loss in efficiency. We use Unix System V, with communication primitives from BSD 4.3. The system has been implemented on a network of Sun-3 and Bull-SPS-7 workstations.

The shared memory primitives of Unix System V are intensively used to support shared objects, as well as the shared tables of the local kernel on a node. An activity which spans several nodes is represented as a collection of Unix processes, one on each node visited by the activity. A job is a collection of activities, together with the tables which represent the context of the job. The persistent object memory has been implemented using raw disk mode.

A detailed description of the Guide implementation on Unix is given in [Decouchant88a]. The Guide language, a prototype of the language designed and implemented by the project, is described in [Krakowiak89].

4 Experience

The experience gained in implementing and developing the Comandos architecture may be summarized under two main headings: experience with building a kernel, and experience in developing applications.

4.1 Experience in developing a distributed kernel

The function of the Comandos kernel is to provide access to a global virtual machine in an efficient, uniform and transparent way. Using the object oriented paradigm, all services and facilities are accessed in this uniform transparent fashion. Reliability and efficiency are however functions that must be provided by the underlying kernel. To this end a number of issues were considered when developing the Comandos kernel on a bare machine. These included granularity of objects, runtime development, access to system objects and system services, IO subsystem, communications and kernel processes. Each of these topics is considered below.

4.1.1 Granularity of objects

In the Comandos Architecture and in TCD design [Cahill88], objects normally were at least a virtual page in size with a header associated with each object. Each object also had to be recorded in the

SS in an LLI (*Low Level Identifier*) map and were referred to by references which contained a hint (usually the virtual address of the header in VOM).

Smaller objects (ie those needing less than a page) could be placed in clusters. In this case, not only did the LLI of a clustered object appear in the SS LLI map, but also in a map at the head of the cluster.

Thus, it could be seen that there would be a large number of objects in the system, with kernel maps being extended constantly to cater for this. Clusters and objects gave rise to the following problems :-

1. Mapping a cluster into VOM implies scanning the cluster map and registering every LLI found there into the VOM per Context Object Table (*COT*). Likewise unmapping a cluster implies doing the reverse.

2. Putting a global object into a cluster involves updating two maps - the logical container map and the cluster map.

3. If a global object is shared between two contexts at the same node, then any pointers (ie virtual address values) it contains must be valid in both contexts. If we consider the VOM Hints - which are such pointer values - then the header of such a global object would have to appear at the same position in both of the virtual memory address spaces. Further, the global object would itself also have appear at the same position in both contexts, because of the shared header and its pointer to the object. This all implies that Oisin would have to organise each virtual address space with a view to how other such address spaces are organised at a node.

Other problems encountered were in the area of the Comandos language. In both the Architecture report and the original TCD design, it was assumed that the Emerald approach [Hutchinson87] could be adopted for the Comandos language with respect to distinguishing local and global objects. That is, language compilers and the Comandos language, in particular, would divide the universe of objects into three: direct (only for basic values like integers); local, and global. The Oisin kernel was to *only* know about globals - locals and directs are to be hidden inside of globals, as far as the kernel is concerned.

The obvious question is whether or not a compiler can actually make this classification at compile time. It turns out to be difficult to satisfactorily separate locals from globals (directs are no problem!), has consequences for compilation speed and hinders separate compilation. Even if the compiler can do it, it will of necessity have to make a conservative judgement about what can be allowed to be a local. Programmer supplied hints might be applicable, but they impose on the programmer, and further may become invalid if the code is re-used. Leaving the issue entirely to compiler technology means we may have more globals than actually may be strictly needed. As a result we may have a fair number of relatively small global objects, exhibiting strong locality of reference and in many cases just having one reference to each of them. Finally, the whole technique is compiler dependent, and mitigates against supporting arbitrary programming languages.

In the final analysis we use clusters of objects. Clusters are uniformly the unit of granularity in the implementation of the Oisin kernel: the kernel thus does not handle individual objects. The consequence of this is that the kernel itself can deal with the issues of naming, location, distribution etc. and be efficient in doing this. The details of the objects and their usage is left to the user mode runtimes and applications.

To be more precise, we define *mature objects* as those which can be referenced from any part of the distributed system. An *immature object* is one which cannot be referenced globally like a mature object, and can thus only be referenced within some limited scope or name space. However an immature object may be *promoted* to a mature object. A *direct object* is a value of a basic type. It is stored completely within a mature or immature object and has no independent existence.

A *cluster* is a collection of immature and mature objects. The cluster defines the limited scope or name space for those immature objects within it. An object in a cluster can refer to other mature objects anywhere in the system, but only to immature objects within its own cluster.

In the end we have developed a system where object invocation between closely related objects in a cluster can be invoked very efficiently (i.e. intra cluster invocation). Also, objects - whether mature or immature - are shareable between address spaces at the same node by means of sharing clusters.

4.1.2 Runtime development

The runtime is a layer immediately above the kernel, executing in user mode, and is primarily responsible for implementing invocation between objects resident in the same address space context. Actually there are two categories of invocation: intra- and inter-cluster. The former is considerably faster than the latter: on our Vax implementation about 2.5 times a null C function call for the former, compared with 25 times for inter-cluster calls. The runtime is also responsible for trapping that an invoked object is absent from its cluster (because it might have been moved into another cluster) or indeed that the entire cluster, ostensibly containing the invoked object, is not (yet) present in the current address space context. The runtime must also detect that the implementation object, containing executable code, for an object maybe currently absent. The runtime manages free space within a cluster, including the creation of new objects and the resizing of existing ones. Finally it implements per cluster garbage collection and supports migration of objects from one cluster to another.

The runtime is intended to be used from a number of application programming languages, and in particular to encapsulate an object written in some language so that it obeys the standard invocation mechanism. The actual encapsulation is typically achieved by pre- processing a compiler input. One of our chief difficulties has been not to enforce any restrictions on register usage of a particular compiled code. In practice each cluster supports only objects written in the same source language, and any critical register usage is indicated to the runtime by the cluster header. So far only Modula-2 and the Comandos language are supported: work is in progress to support in particular C++.

Multiple lightweight threads (*activities*) may be simultaneously present in an address space context, which complicates the runtime and invocation mechanisms because one activity may cause garbage collection to occur, or may move an object, while another activity is currently building an invocation frame on its stack, or using an object which is consequently relocated. As in Trellis/Owl [Moss87] we are concerned to make certain instruction sequences atomic, and to do so without incurring significant overheads. The actual invocation is atomic due to a combination of restartable code and knowledge of certain register usage.

Critical sections within the runtime itself are made atomic by using spin-locks and/or longer term locks involving kernel notifications.

Garbage collection is currently done on a per cluster basis, and currently using a simple mark-sweep algorithm. It requires close kernel-runtime interaction. Activities currently executing within the cluster (ie. with SELF or "this" in the cluster) are suspended during the collection, and their stacks are also examined for object references into the cluster. Other activities, not using the cluster, are however allowed to continue their execution unless they attempt to invoke into, or return from an invocation into, the cluster. If a cluster is shared between address spaces, these same mechanisms must be used, and notably the stacks from activities in different contexts must all be scanned for appropriate object references. Our experience has been that this has considerably complicated the implementation effort, and one which makes us query the utility of shared objects between address spaces in the Comandos model. We did consider preventing the creation of new objects, and resizing of objects, in such shared clusters, but this seemed overly restrictive particularly since use of a shared cluster is often transparent to applications.

Migration of an object between clusters has been relatively simple to implement, provided that the object is not currently SELF to some activity. However if this is the case, care is required to migrate the activity as well (perhaps to another node if the target cluster is remote), and migrating SELF is a particular case. Currently migration of activities in this way is not operational.

A version of the runtime layer has been ported on top of Unix (currently Ultrix V3.0). Crucially, compiled images will run without any modification on top of the Unix version or on top of the native kernel. An individual cluster is stored and loaded as a Unix file and thus, as in the native kernel, a large group of objects can be mapped in a single step. However the Unix version does not yet support lightweight threads, distributed invocation or shared address space clusters. Nevertheless the Unix version is sufficient to aid debugging of many applications since distribution (but not concurrency) is largely transparent in the Comandos model.

4.1.3 Access to system objects and system services

Early on in the development the need for access to kernel facilities was seen. Simple things such as character i/o needed to be approached in a coherent fashion with regard to user applications. Thus, it was considered to use the object model and define a number of system objects with operations. For all objects which the runtime cannot locate in the address space an *AbsentObject* system call is made on the kernel. In the case of ordinary objects the kernel ensures that the appropriate cluster is mapped in the address space or that the invocation takes place in the address space at a remote node. In the case of system objects these are trapped by the kernel and the operation carried out locally in the kernel space. To the runtime it appears that these operation invocations take place remotely. Thus all services appear as objects in the model.

In the case of non-kernel, system support services - i.e. facilities that are provided by external servers in classical "object based" distributed kernels - a similar mechanism is needed. A major issue is that of where such objects should reside and also how they can be protected (i.e. from malicious users). An early example in Comandos is the Name Service: currently this is programmed as individual directory objects. If any user accesses a directory, it is mapped (currently, completely unprotected) into his address space context. If two users simultaneously access the same directory, their jobs diffuse if necessary to a common node, and the directory is simultaneously mapped into both contexts.

Some time ago, we believed that some form of inter-job invocations would be necessary so as to overcome the problems of shared and protected access to service objects. This however was not generally received well as it seemed to be against the general model of Comandos.

Our current proposals are to instead extend the Comandos computational model so that a job can have several address space contexts *at the same nodes*: the current model is only one space per job at a node. In this way protected objects can be safely mapped into their own address space, and yet used by "clients" jobs.

For example, a Name Service directory could always be mapped into a protected context when it is invoked by an ordinary client. The client can only invoke the operations exported by the Name Service interface; further while executing the operation methods, the Name Service is guaranteed that the client cannot compromise the directory (assuming that the Name Service implements its own operations correctly). In particular, an asynchronous parallel activity launched by the client cannot interfere. Further, the protection can be mutual: the client need not trust the Name Sevice, and the Name Service will be unable to directly access or corrupt any other objects owned and used by the client.

Separating objects which mutually distrust one another allows us to use protection based on operations, rather than just on pure read/write access. It also possibly reduces the need to support objects shared between jobs (ie shared memory segments between address space contexts).

4.1.4 I/O and Network subsystems

Experience in the development of Oisin, and a background in Unix, led us to believe that some work in structuring device drivers would be fruitful. The main objectives which we had were that the software modules which control the hardware devices should be well organised and have well defined, clean interfaces; that the i/o subsystem should exploit any available hardware parallelism; that hardware configurations with multiple access paths should be supported; and that configuration of the i/o subsystem should be determined at boot time and re-configurable, if desired, while the system is running.

The i/o subsystem in Oisin is based on i/o paths with device objects. There is an overall device manager, which maintains the current device configuration. The device manager also controls the routing of i/o request and completion notification packets up and down various i/o paths. Each component in the path is called via a software interrupt mechanism.

A number of Unix device drivers have been ported into the Oisin i/o system with reasonable ease.

The modular approach of the i/o subsystem led us to believe that many devices could be attached and reconfigured in the system. To this end it was decided to develop the communications subsystem as a number of pseudo devices. In this way, processing at the different levels in the network stacks (eg protocol drivers, and routers) could be handled by different i/o components on i/o paths registered in the device manager.

Lightweight, kernel mode, processes are used to handle certain kernel events. In particular, incoming invocation requests from remote nodes are treated in this way, after ascending their respective network protocol stack and emerging from the i/o subsystem.

The basic protocol used between kernels is the *Inter-Kernel-Messaging* protocol. This is a resonably conventional RPC protocol: we have implemented it on raw Ethernet, and are currently extending support to include IP and other transport mechanisms. The i/o and communication systems are also currently being critically evaluated.

4.2 Experience in developing distributed applications

This section presents an evaluation of the Comandos architecture from the user's point of view, as to its suitability to support distributed applications. This evaluation is essentially based on the experience gained on the Guide prototype.

Experimental applications have been developed on the Guide prototype since mid-1988. The aim of these experiments was to test the main mechanisms of the system, with emphasis on the following aspects:

1. the integration of the components of the system: compiler and run-time support, system services, kernel, virtual memory manager, secondary storage, communications.
2. the suitability of the Comandos architecture for the programming of distributed applications.

In the next sections, we give an overview of the main experiments and draw some preliminary conclusions.

4.2.1 Integrating existing applications

Two large scale applications have been integrated into Guide: the X-Window server, which provides window management suuport for applications, and Grif, a high-level editor for structured documents. We briefly describe how this integration has been achieved.

- Integrating X-Window. We defined a new object type, X_Server. The interface of this type includes the primitives of the X-Window server, and a high-level library (string management, etc) developed for our own applications. This type is implemented by a type implementation (also called X-Server) which encapsulates the program of the X server and the library. In order to use a X-Window primitive in a Guide application, a user must create a new instance of X-Server. The X-Window and library primitives are available as calls to methods of this new object. The executable code of these methods is shared by all the instances of X-Server on a given node.

- Integrating Grif, an advanced document editor. We defined a new object type, GrifDocument, which is the type of the documents handled by the Grif editor. Calling the Edit method on this object starts an editing session. The user may now create a new document, which will be stored in the system as a Guide object, or open an existing document. If this document is in the original Grif format, it is automatically converted to a Guide object.

This first experience shows that the integration of an existing application into Guide can easily be done in a matter of days. The current integration technique is primitive and essentially amounts to a coarse-grained encapsulation. The next step is to achieve a closer integration, which is costlier since it involve a partial rewrite of the applications in order to use the object model in their internal structure.

4.2.2 Developing new applications

Mail Electronic mail is often regarded as "the" typical distributed application. As such, it is frequently used as an illustration for distributed systems structuring.

Programming this application in the Guide language involves a structure different from that of most traditional mail service implementations. A salient feature of this structure is that distribution never explicitly appears, which results from the basic decision of providing a transparent structure. Actually, the objects used in the mail service (lists of clients, mailboxes, messages, etc) may be distributed on several nodes, but this distribution is not imposed by the structure of the mail program (e.g. the messages contained in a mailbox need not be on the same node, etc). The distribution may be governed by considerations of efficiency, security, availability, performance, etc.

The application is organized around the following object types (we only consider here the services available to the clients):

messages (Type Message). Objects of this type are the messages that are sent and received by clients.

mailboxes (Type Mailbox). Objects of this type are essentially implemented as lists of references to messages.

directories (Type Mail_Directory). Objects of this type implement the mapping between a list of client names and the mailboxes associated with them.

mail service (Type Mailer). Objects of this type implement the interface provided to the clients.

When a client calls the electronic mail service, a mailer object (an instance of type Mailer) is created and lives until the end of this client's session . This object is created on the node when the mail service was called. Other objects such as mailboxes, messages, etc, may be created on other nodes. Clients are not aware of the distribution of these objects.

A simple bibliography database This application illustrates the problems of accessing shared objects in a distributed environment. In addition, it illustrates the capabilities of the inheritance mechanism provided by the type and implementation hierarchy. Different kinds of references such as Book, JournalArticle, TechReport, etc, are defined by types, each of which

has a specific format. Subtyping allows to share common format fields and common methods between types. A new type is easily defined as an extension of an existing type.

Persistence is another important feature illustrated by this application. An object has a permanent existence in the system after it has been created by a New operaton. There is no need, therefore, to store it into a file system to retrieve it at later time. References allows a system-wide, location-independent internal naming for persistent objects. A bibliography database may be distributed on several nodes, but it provides a uniform interface to its users.

4.2.3 Distributing an existing application

All applications have been initially programmed and debugged on a single node and thereafter ported to the distributed system. This experience has been positive. Virtually no additional work was necessary. One interesting application is the distributed Hanoi tower game. This is the only application where distribution explicitly appears. Starting from the single-node version, it was easy to develop a distributed version where the three towers may be located on different nodes. The location is interactively requested from the user when the program starts, and the program uses the Create primitive which allows to specify the container where an object is created. The main body of the program was unchanged. The method calls which were performed locally in the single node version, are interpreted as remote if the location algorithm finds that the called object is on a distant node. This is totally transparent to the user.

5 Conclusions

The first conclusions that may be drawn from this preliminary experience are as follows :

Object model and language The experience using an object-oriented language supporting persistent objects to program distributed applications was positive. A learning phase was necessary to adapt to the specificities of object-oriented programming. As many authors have noted, programming with objects involves a new style of design and programming. This is adequately illustrated by the mail application example. The main design tool is no longer the familiar hierarchical decomposition, but a decomposition into types, where the aim is to regroup logically related functions into a type and to reuse existing implementation types, either as call or by extension. This in turn is a strong incentive for the design of reusable implementation types.

Persistent objects Using persistent objects frees the programmer from the need of explicit object saving. The counterpart is the cluttering of object storage by garbage. A distributed garbage collection algorithm is currently been designed, but has not yet been implemented. Objects are explicitly destroyed in the current version of the system.

Performance problems (due to the overhead of oid dereferencing and dynamic binding) are a well known drawback of persistent object systems. In order to overcome these problems, Guide has proposed a mechanism based on internal objects. Using internal objects improves performance but the applicability of these objects is limited. In addition, in the current version of the Guide language, internal and persistent objects are not equivalent. In order to convert a program written with internal objects to a program using only persistent objects, changing the declarations is not enough, and the program of some methods also need to be modified. This limitation should be removed in a future version of the language.

In the Oisin kernel, similar problems have been approached in a language independent fashion via clusters. In our view, the use of clusters has been successful, since loading and unloading a large group of objects becomes relatively trivial and reasonably efficient. Further, configuring clusters can be independent of compiled code, assisting re-use of code in situations for which assumptions made at coding time may no longer be reasonable.

Object location and execution transparency A consequence of execution transparency has been to blur the difference between centralized and distributed applications. The task of locating and binding objects is transferred to a primitive of the system. This simplifies the implementation of applications, since an application may be developed and debugged on a single node before being ported to the network. It is still possible to explicitly locate objects on specific nodes if required by the application.

Overall, we have found that implementing a computational model involving mixed language programming, object invocation, transparent persistence and transparent remote invocation, in a multi-user environment, to be extremely challenging. A large number of difficult implemention decisions have been made. Our chief effort at this time is stabilising our implementation and critically reviewing it.

6 Acknowledgements

Within Bull/LGI, we would gatefully acknowledge Roland Balter, Marie Meysembourg, Eric Paire, Cecile Roisin, Xavier Rousset de Pina, and Gerard Vandome.

Within TCD, we would greatly acknowledge the work and input of Sean Baker, Vinny Cahill, Donal Daly, Alexis Donnelly, Ahmed El-Habbash, Andre Kramer, Damien Lynch, Maurice Martin, Faris Naji, Annrai O'Toole, John Slattery, Gradimir Starovic, Brendan Tangney, Bridget Walsh and Iseult White.

We are also conscious of our indebtedness to our many colleages outside of LGI, Bull and TCD with whom we have collaborated in Esprit Comandos. We would in particular express our appreciation to Jose Alves Marques, Elisa Bertino, Richard Cooper, Paulo Guedes, David Harper, Helmut Meitner, Andreas Ness, Paulo Pinto, Rosalee Pocock and Andrew Redman.

References

[Cahill88] Cahill V.J., "OISIN: The Design of a distributed Objetc-Oriented Kernel for Comandos", M.Sc. thesis, *Dept. of Computer Science, Trinity College Dublin*, Mar 1988.

[Decouchant88a] D.Decouchant, A.Duda, A.Freyssinet, E.Paire, M.Riveill, X.Rousset de Pina, G.Vandome, "Guide:an implementation of the Comandos object oriented architecture on Unix", *Proc. EUUG Autumn Conference*, Lisbon, oct 1988, pp. 181-193

[Decouchant88b] Decouchant D., Krakowiak, S., Meysembourg M., Riveill M., Roisin, C., "A synchronisation mechanism for typed objects in a distributed system", *Workshop on Object-Oriented Concurrent Programming*, San Diego, Oct 88 (in SIGPLAN Notices, 1989)

[Horn87] Horn C.J., Krakowiak S., "An object-oriented architecture for distributed office systems", *Proc. ESPRIT Technical Conference*, Brussels, 1987 North Holland.

[Horn89] Horn C. and Donnelly A. "Architectural Aspects of the Comandos Platform", *Proceedings second international workshop on Distribution and Objects*, Karlsruhe, April 1989, sponsored by Decus.

[Hutchinson87] Hutchinson N. C., "Emerald: An Object based language for distributed programming", PhD Thesis, *University of Washington*, 1987.

[Krakowiak89] Krakowiak S., Meysembourg M., Nguyen Van H., Riveill M., Roisin C., "Design and implementation of an object-oriented, strongly typed language for distributed applications", submitted to *Journal of Object-Oriented Programming*

[Leach83] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, B. Stumpf "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communications*, pp843-857, Vol SAC-1, No 5, November 1983.

[Marques88] J. Alves Marques, R. Balter, V. Cahill, P. Guedes, N. Harris, C. Horn, S. Krakowiak, A. Kramer, J. Slattery, and G. Vandome, "Implementing the Comandos Architecture", *Esprit'88: Putting the Technology to Use*, pp1140-1157, 1988 North-Holland.

[Moss87] Moss J.E.B., Kohler W.H., "Concurrency Features for the Trellis/Owl Language", *ECOOP '87*, Palais des Congres, Paris, France, June 1987.

Prototyping a distributed object-oriented operating system on Unix

Marc Shapiro
INRIA, BP 105, 78153 Rocquencourt, France; shapiro@sor.inria.fr

August 1989

Abstract

The SOR group at INRIA has built a prototype distributed object-oriented operating system, called SOS, on top of Unix. SOS is based on migratable medium-grained elementary-objects, on top of which all its other basic mechanisms (such as composite objects, dynamic linking, and dynamic type-checking) are built.

SOS supports distributed or "fragmented" objects. A fragmented object is created by spreading out *proxies* from a *provider*. The public interface of the fragmented object is provided locally by proxies. Proxies may communicate directly, without going through the public interface (via messages, sharing, or any other means).

The most positive accomplishment of SOS is its elementary-object concept. A programmer-defined object can impose its own semantics or policies on system-implemented mechanisms, thanks to our *upcall* and *prerequisite* mechanisms. For instance object migration and storage are performed under the control of the system, but they respect the semantics of each individual object.

One negative aspect is that a fragmented object can only be created dynamically. Other problems arise from the protoyping environment of Unix and C++.

1 Introduction

The object-oriented programming methodology is becoming increasingly popular, for all sorts of applications. Many object-oriented programming languages exist, such as Smalltalk [9], C++ [24], Eiffel [15], CLOS [7], etc. Each compiler enforces its own object model, and deals with the inadequacies of existing operating systems in its own way.

The goal of the SOR group of INRIA is to implement an object management support layer common to all applications and languages. This should: facilitate the implementation of object-oriented language compilers; make applications more efficient; allow independent applications to communicate and share objects, without prior arrangement.

The services of the common object management support layer include support for creating, deleting, migrating, storing, localizing, and invoking objects. If these services are sufficiently complete, low-level, generic, language-independent, application-independent, and efficient, then they can legitimately be called an *object-oriented operating system.*

Within the office-workstation Esprit project SOMIW (1985–1988) we have built a prototype called SOS. It has been used for the SOMIW applications, such as BFIR2, a multimedia document toolbox, and Images, an user-interface management system. SOS is written in C++ and prototyped on top of Unix.

SOS supports an elementary-object model which is both simple and powerful. A reasonable granularity is of the order of a hundred bytes and up per object. Composite objects, object storage, dynamic linking and dynamic type-checking are built on top of elementary-object mechanisms.

In addition, SOS extends the object concept to distributed, or "fragmented", objects. The public interface of the fragmented object is provided locally by its fragments, which are the elementary objects. This encourages the structured design of distributed applications based on the "Proxy Principle" [19]. All the SOS system services (for instance, the Name Service [11]) are built as fragmented objects with local proxy interfaces.

We now have accumulated enough experience to assess the SOS design and implementation. A most positive aspect is its elementary-object model. Fragmented objects have proved a good way to structure distributed applications, although they are hard to use. A weakness of our implementation is that fragmented objects cannot be static or persistent. The proxy concept poses a protection problem. The implementation of SOS is not really language-independent. Object migration imposes restrictions on the use of C++ as the application programming language. The prototype is slow.

In the remainder of this paper we will explain some of the aspects of the SOS prototype. We expose design rationales and implementation, and discuss features and limitations. We start with a short comparison with similar work, in section 2. Section 3 first gives some background on SOS concepts and implementation. Section 4 is about elementary objects. It is followed by section 5, an explanation of fragmented objects. Follows section 6, which discusses object migration. Finally, in section 7, we give an assessment of the design and implementation of the prototype.

2 Comparison with similar work

Emerald is an object-oriented language for distributed programming, featuring fine-grained mobillity [10]. The compiler transforms the user-defined object representation in order to facilitate migration: its first few bytes are a standard descriptor, and all fields of a similar type are grouped together. Conceptually, all objects live in a single, network-wide address space. An object reference is global, but a local reference is optimized into a pointer.

In contrast, the SOS approach is operating-system based. We do not assume any standard representation. Instead, system information is well separated from programmer-defined data, and the system performs upcalls on objects. Instead of a single address space, we stress *structuring* the universe.

Choices [3] is a family of operating systems built using object-oriented design. The services it exports to applications are fairly conventional. The emphasis in SOS was not its internal design, but providing new services to facilitate the implementation of distributed object-based applications.

Clouds [5] is another object-oriented OS. Its emphasis is on integrating support for reliable objects in the low level of the system. (Our current design has no particular provisions for reliability.) Their objects are presumably much larger-grained than ours, since a Clouds object executes in its own address space.

Guide/Comandos [6] is a language-driven distributed programming environment. The universe is structured in separate, multi-machine address spaces called domains. When a domain needs access to an object located on a remote machine, it extends itself to that machine, and maps the object in. This structure is easier to use than SOS's fragmented objects; however the latter scales better, and deals better with replication.

Gothic [1], a language and system for reliable distributed programs, is based on a theory of fragmented objects invoked via "multi-functions" (side-effect free invocations, with co-ordinated multiple threads), supported by the language. Our fragmented objects are ad-hoc but more flexible.

3 Background

3.1 SOS concepts

SOS is an object-oriented operating system. It provides support for arbitrary user-defined objects, including object creation, destruction, migration, storage, localization, communication, naming, etc.

An *elementary object* is an user-defined data segment with a system descrip-

tor.[1] Considering the descriptor overhead, a reasonable granularity for the data segment is a size of 50 or 100 bytes and up.

We assume that the data was created using an object-oriented programming language compiler, and that it is accessed only via its type-checked procedural interface. An object accesses system services by calling the appropriate primitives; we call this a *downcall*. Conversely, the system can invoke, with an *upcall*, a few well-known procedures of an object. For instance, a cross-context invocation (see below) is executed by upcalling the stub procedure of the target object; each object has its own stub which can be redefined at will.

An object is designated by its address (within the context), or globally by a *reference* containing an OID (object identifier) and a location hint.

SOS comprises a kernel and system services running on top of it. The kernel provides separate address spaces (*contexts*), light-weight threads in a context (*tasks*), and inter-context communication. A context may contain any number of elementary objects. Elementary objects may migrate between contexts; at any point in time, an elementary object is active within a single context, or stored on disk.

SOS extends the object concept to distributed or *fragmented* objects (see figure 1). A fragmented object is implemented as a *group* of elementary objects located in different contexts; i.e. its representation is the reunion of the local "fragments".

Just as an elementary object can access its own representation, bypassing the procedural interface, similarly the individual fragments are allowed to use untyped communication to each other: cross-context invocation, communication protocols, shared memory, shared files, etc. Objects which are not fragments of a same group are not permitted to communicate in this manner.

A fragment may create and add a new fragment to the group, and export it to another context. Group membership is preserved across migration; thus the group grows by spreading.

Applications on SOS are designed according to the "Proxy Principle" [19]: to use some service, a client invokes a local *proxy* for the service, i.e. a local object which is a fragment of the group implementing that service. If such a proxy is not locally available, it must first be acquired by sending an import request to a *provider* for that group, i.e. a particular object in charge of delivering proxies.

For instance, for a graphics program to output to the screen, it will request a window proxy; the window manager is the provider for this resource. The window manager will reserve the window and allocate it to a window server object; it will create a window proxy which is exported to the graphics program.

[1] Composite objects, with multiple data segments connected by pointers, are built on top of elementary objects, but they will not be considered here. Similarly, object storage is built on top of migration. See [21].

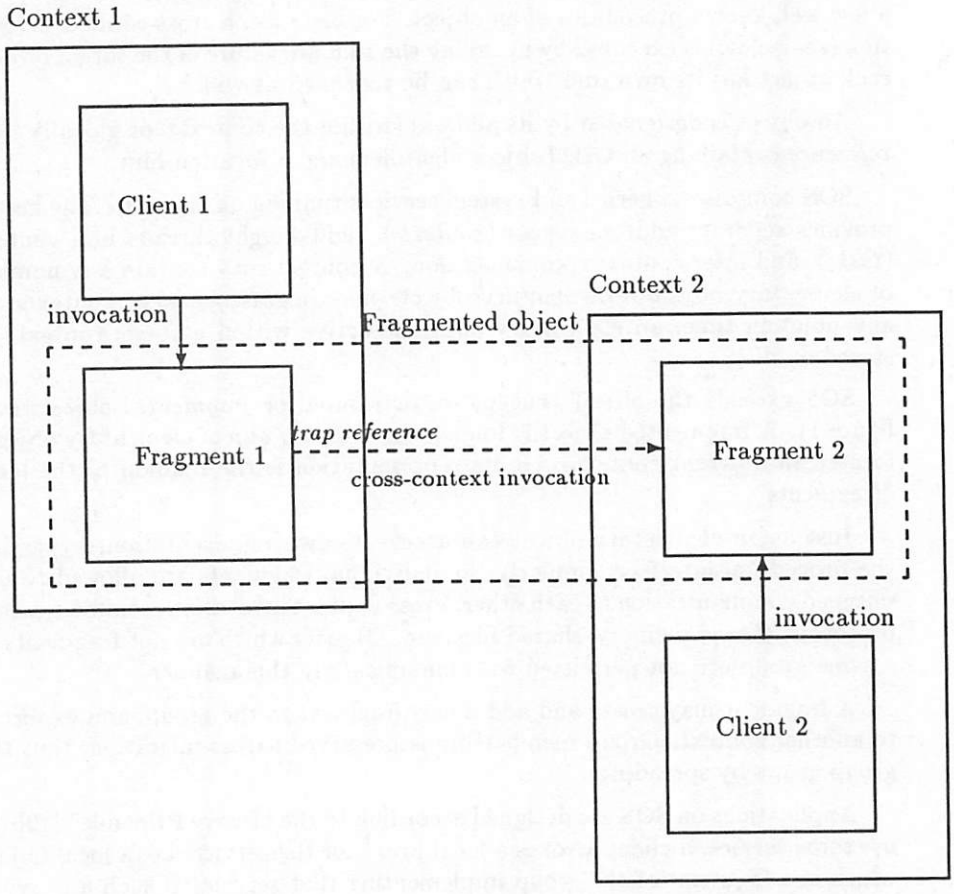


Figure 1: A fragmented object (group) with two fragments, used by two clients in different contexts.

The proxy has a graphical interface (e.g. drawVector, displayCharacter, etc.). It may interpret some requests locally (e.g. getWindowSize); others will be buffered before being sent down a *channel* to the window server when appropriate.

In this article, we are only interested in the objects managed by SOS, which we will call *SOS objects*. Only objects which are intended to be migrated, stored, or remotely accessed need to be SOS objects; application programs are free to manage their other, internal, "plain" objects. In the remainder of this paper we use the word "object" for "SOS object".

3.2 The prototype

Our prototype is implemented in C++ on top of Unix (SunOS 3.4). This article describes SOS Prototype Version 4, which was delivered to the SOMIW partners in the Fall of 1988 [22, 23].

SOS objects are instances of the predefined class sosObject (or of a compatible class). C++ has so-called virtual procedures, invoked indirectly via a table of procedures [8], itself accessed by a pointer in the object's data. A class may override the pre-defined actions of a procedure, by replacing the corresponding entry in the procedure table. Upcalls are performed via this table; unfortunately, this is not language-independent.

Separate address spaces are provided by Unix. Tasks are implemented as a library (the task library of C++ [25] with some additions). Context management is performed by a Unix process called sos. Inter-context communication uses Unix-domain stream sockets.

On each machine, sos automatically starts a number of system-service contexts, the local servers for: the Acquaintance Service (AS) or object manager, the Storage Service, the Name Service, and the Communication Service (CS). Each new context starts with a pre-installed proxy for the AS, which allows the application to import proxies of the other (system- or user-defined) services. Applications are run from the Unix shell or the debugger.

In the remainder of this paper, we will take a look at the design and implementation of the SOS prototype, and evaluate it in the light of our experience. We will concentrate on the aspects of distributed management and communication of objects; for a more in-depth introduction to SOS, see [20, 22]. With one exception (export, section 6.2.1), all the interfaces given here are implemented with the explained semantics. They have been in use in the prototype for at least one year.

4 Elementary objects

The basic entity managed by the SOS Acquaintance Service (i.e. the object manager) is the *elementary object*. We have made the elementary object as simple as possible, a "least common denominator" for all uses.

At any point in time, an elementary object exists in a single context on a single machine. Each elementary object is different from all others; it is characterized by its own unique identifier called its *concrete OID*. An elementary object is known to SOS by its descriptor, called Acquaintance Descriptor (AD). There is a table of AD's per context, managed by the context's AS proxy.

An AD for some object contains the following information (the items in italics have to do with migration and groups, and will be defined later):

- Its concrete OID and (possibly) a list of *group OID*'s,
- The reference of its code object and (possibly) a list of *prerequisites*,
- The address and size of its data segment,
- (Possibly) Its list of *trap references*.

The class code is a predefined class of elementary objects. A code instance holds the compiled code for some class. For instance the code for some user-defined class X is managed by the code instance code_for_X. The reference from the AD to the object's code is necessary for migration.

The following table[2] gives the downcall interface for elementary-object management. There are no upcalls.

Downcalls for elementary-object management	
(sosObject *constructor*)	Object creation
(sosObject *destructor*)	Object destruction
obj . setCodeRef (ref)	Set code reference of object
find (ref1, radius) → ref2	Search for object location
getAddress (ref) → obj	Translate global reference to local address
getReference (obj, OID) → ref	Translate address to global reference

4.1 Creation and destruction of elementary objects

In C++, creating an object triggers a chain of calls to *constructor* procedures, starting from from the actual implementation class, up to the root of the inheritence tree (in this case, sosObject), and back down to the implementation class.

[2] The pseudocode

a . b (c) → d

means: invoke procedure b of object a, with in argument c, and returning value d. If no object a is mentioned, then the procedure is a primitive (actually, a procedure of the kernel or of the object manager).

A constructor is a mix of compiler-generated and user-defined code. Memory for the object is allocated (by malloc) in the compiler-generated part of the implementation class constructor, and passed up as a parameter to the sosObject constructor.

Thus, there is no explicit primitive for object creation: it is subsumed by the the sosObject constructor. It allocates an unused AD, and fills it with a newly-allocated OID, and with the address and size of the data.

The size of the data is not explicitly available to the sosObject constructor: it is taken from the malloc header. The reference to the code object is not available either; the constructor sets it to nil. The other parts of the AD are also initialized to nil.

The implicit AS interface for object deletion is the *destructor* procedure of sosObject, called automatically when an instance is deleted. The destruction of a context or a processor crash deletes all the contained instances. We are currently designing a mechanism to propagate object-destruction events to dependents of an object.

4.2 Assessment

The rationale of the above design is that the object creation and destruction primitives are rendered transparent by the C++ inheritence, thus simplifying the task of the application programmer. One drawback is that the system interface is not clearly identified and not language-independent.

Another problem is that the sosObject constructor does not have all the necessary information; for instance the size of the data is obtained by the malloc-header hack. Similarly, the code reference can not be set by the constructor; a separate call to setCodeRef is necessary prior to migration.

4.3 Other primitives for elementary-object management

The find procedure of the AS, given a reference to an object, finds the actual location of the object (possibly by asking all the AS proxies within the specified radius), and returns a reference containing that exact location. If the argument is a reference to a group (see below), the returned value is the reference of its closest fragment.

getReference and getAddress translate between local object addresses and global references. If getAddress is passed a reference of a group, it returns the address of its local proxy, if any. The OID argument of getReference allows to pick between a reference to the elementary object itself, or to its group.

5 Fragmented objects (groups)

A fragmented object is implementented as a *group* of elementary objects, called its fragments. (Or, alternatively, a group is a single object with a fragmented representation.) Clients of the group may access it locally by its strongly-typed procedural interface, provided by the proxy fragments; its public interface is a sort of "union" of its fragments'.

Each fragment of the group may access the object's internal representation; fragments may communicate via untyped shared memory or messages, for instance.

The group is conceptually a protection domain, entered by invoking a local proxy.

This is illustrated in figure 1.

The following table shows the interfaces for group management.

Group downcall interface	
addGroupOID (obj, OID) → index	Create a new group
obj1 . giveMyOID (obj2, index1)	Put obj2 in same group as obj1
obj1 . setTrapRef (obj2, opaque) → index1	Establish channel from obj1 to obj2
obj1 . giveTrapRef (obj2, index1, opaque) → index2	Duplicate channel
crossInvoke (callMsg, segs, index) → replyMsg	Send invocation, receive reply on channel
Group upcall interface	
stub (callMsg, segs) → replyMsg	Receive invocation, return reply

A group is characterized by the fact that each fragment carries the OID of the group, in addition to its concrete OID. An elementary object can be a fragment of zero, one, or more groups.

Members of a group enjoy mutual communication privileges, which are denied to non-fragments. An invocation *channel* is a unidirectional connection between two elementary objects on the same machine, materialized by a *trap reference* in the source object's AD pointing to the target.

Other types of communication within the group, such as shared files, are also available, but will not be detailed here. Shared memory should be possible, but we never implemented the appropriate interfaces.

5.1 Group management

The primitive addGroupOID assign a fresh group OID to an object, in order to start a new group.

A group is created implicitly by giveMyOID, which gives away an existing concrete or group OID (designated by its index in the list of OID's of obj1), to some object.

A group is destroyed when its last fragment goes away.

Channels are created by the primitives setTrapRef and giveTrapRef. The former procedure creates a channel between the current object and the first argument; it returns the index of the channel in the current object's trap reference list.

giveTrapRef duplicates an existing channel: before the call, obj1 has a channel at index index1 to some receiver; after the call, obj2 also has a channel to the same receiver, at index index2.

As its name implies, the opaque argument is not interpreted by the system. It is simply stored at the sender end of the channel, and will be automatically prepended to every invocation sent on it. The receiver may test the opaque field of remote invocations to distinguish between its callers, and test their access rights.[3]

5.2 Cross-context invocation

The primitive crossInvoke sends an invocation on a channel of an elementary object, and returns a reply.

The arguments to crossInvoke are (in addition to the channel index) a message, and possibly a list of segment access rights. The message is of limited size (1024 bytes); any larger data is to be passed as a segment right.[4] Available rights are read, write, and create.

A cross-invocation causes an upcall to the stub procedure of the receiver within a newly-allocated task. The receiver gets a copy of the invocation message, and may access the segments according to the rights passed. stub returns a return message which is copied back to the caller. This procedure plays the role of Nelson's "client stub" [2].

5.3 Assessment

5.3.1 Constructing a group

Currently each fragment type is programmed "by hand", and there is no guarantee of consistency even within a particular type of group. We are working on a new tool, a "fragment generator" (similar to an RPC stub generator). It

[3]The opaque attribute of a channel is similar in concept to the rights field of a capability in Amoeba [16] or Chorus [17].

[4]This is modeled after the V-System RPC [4].

will take care of the common aspects of programming fragments and providers (viz. allocating group OID's, setting up channels, allocating message buffers, etc.). It will also allow to define group types with a well-defined structure, and enforce their internal consistency at compile time. Finally, it will provide help in coordinating state changes between fragments.

5.3.2 Protection

Only the currently-executing elementary object should have access to its own channels. The kernel attempts to enforce this, taking advantage of the fact that the C++ compiler adds a hidden argument to all invocations, which is the address of the invoked object. When crossInvoke is executed, the kernel gets the first argument of the penultimate stack frame, and considers it to be the current object (if it is a valid object address). The index argument is relative to that object.

Getting the current elementary object from the stack is a weak way of enforcing the group protection domain at run-time. Weak enforcement is acceptable, because groups are intended as a program structuring concept, not a confidentiality mechanism.

Given our environment (Unix and standard hardware), and the granularity of objects, it was unfeasible to implement a stronger form of run-time protection at a reasonable cost. A structured memory organization, like that offered by capability machines, might improve run-time protection. A more attractive idea is to enforce the integrity of the group at compile time. Our proposed "fragment generator" is a step in this direction.

To provide some protection of the group against spurious membership, giveMyOID and setTrapRef can only connect objects within the same context. The normal way of creating a group is to create proxies locally and migrate them (see below) to another context; group membership and channels are preserved across migration.

5.3.3 Static groups

There is also a need for static groups. For instance, a system service such as the AS, the Name Service, the Storage Service, or the Communication Service, is implemented by one server on each machine, which comes up at boot time. In order to communicate with its remote peers, it must already be a fragment of their group as soon it starts up. Currently, a protection loophole is needed to circumvent this problem: when the server comes up, it forges an OID with a given value (taken from a configuration file) and inserts itself in the group using addGroupOID.

This loophole should be protected by some privilege but in fact it is not. Better still, both the group and its fragments (the servers) should be *persistent*. SOS supports persistent objects, as a service above the basic mechanisms described here; a much tighter integration is needed to support persistent groups.

5.4 Communication protocols

The only communication protocol implemented by the kernel is a cross-context invocation along a trap-reference channel, within the same machine.

Remote (across machines) access and other protocols are performed by *protocol objects*, implemented by the Communication Service [12, 13, 14]. The CS offers a library of protocol types, such as multicast and stream protocols.

A protocol object is layered underneath the application object which it serves; this is illustrated by figure 2. A protocol object is itself a group of cooperating elementary protocol objects, instantiated in the communication service contexts of the individual machines. A channel is made to use a protocol by setting the trap reference to point to the appropriate elementary protocol object.

An elementary protocol object has two privileges: it can access the network, in order to implement remote communication; and it can be the source or the target of a trap reference, even though it is not a fragment of the application object. In all other respects, a communication object is a standard group.

6 Migration of elementary objects

A client gets access to a new service by getting a proxy for the service: the corresponding group migrates a fragment into the client's context. Migration is completely generic, thanks to appropriate upcalls (givePproxy starts an importation; the *reinitializer* finalizes a migration), and to the code and prerequisite objects.

Before explaining the migration interface, let us look at the migration algorithm.

6.1 Migration algorithm

Suppose elementary object X is to be migrated from source context S to destination context D. The algorithm starts when the decision to migrate X has been notified, and all access rights have been checked; we will ignore error cases. The algorithm is the following.

1. Make X unavailable to users in S.

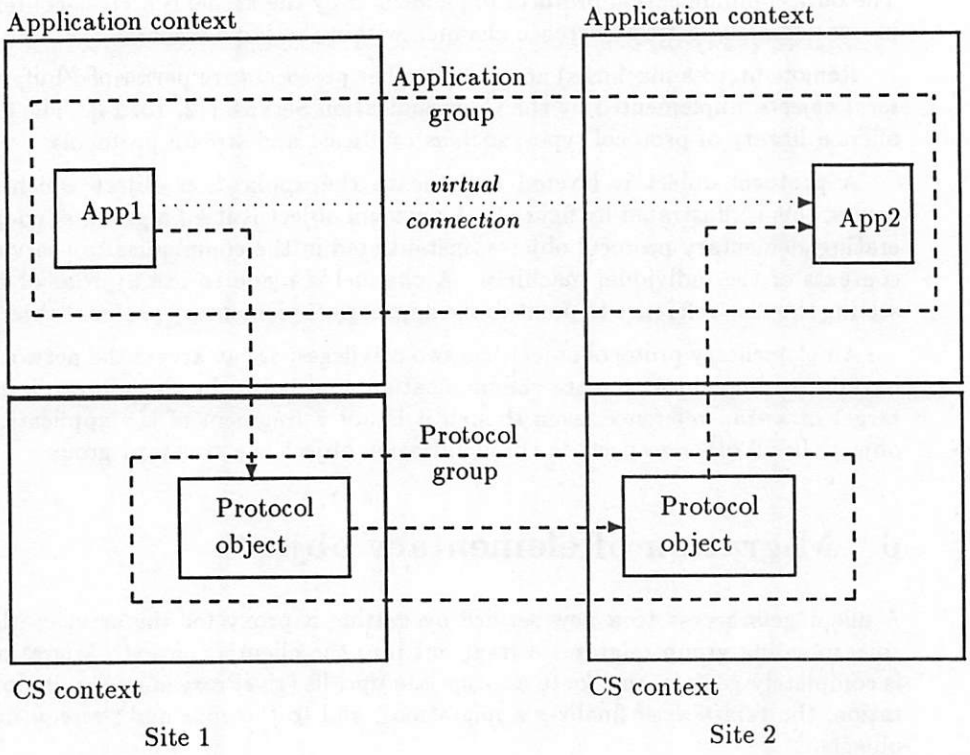


Figure 2: Fragmented protocol object, layered underneath the fragmented application object which it serves. The protocol object has the privilege of setting trap references which cross a group boundary.

2. Copy the AD of X from source to destination context. All the contents of the AD are preserved: its concrete OID, group OID's, trap references, code reference, prerequisite references, and size of data segment. However the address-of-data-segment field is invalid.

3. Using information in the AD, copy X's data from S to some arbitrary free location in D. Update the data address field in the destination AD.

4. If (a proxy of) X's code object is not yet present in D, import one. If present, skip this step. Similarly, import all prerequisites of X, if not already present.

5. Upcall the *re-initialization* procedure of X in D.

6. Make X available to users in D. The data and AD of X are destroyed in S.

7. The trap references of X have become invalid in D. The first time an invalidated trap reference is used for cross-invocation, the kernel executes special code to revalidate it, possibly in cooperation with the Communication Service.

The above describes the "move" variant of migration. The "copy" variant differs slightly: a new concrete OID is allocated in D (step 2), and the source copy is not destroyed, but instead is made available again (step 6).

6.2 Migration interface

The migration interface is given in the following table.

Migration downcall interface	
sosImport (key, importReq, "class", provider) → obj, procTable	Request import of obj of type class
new dynamic (provider) class (importReq, ...) → obj	Same, from C++ programs
obj2 . export (desc, index2)	Export described object along channel of obj2
obj . giveSelf () → desc	Use "move" semantics for migration of obj
obj . giveCopy () → desc	Use "copy" semantics for migration of obj
Migration upcall interface	
obj1 . giveProxy (importReq) → desc	import request for object described by desc
(re-initialization)	Finalize migration

There are two possibilities for migration: import and export. We will start with export, which is the simpler of the two.

6.2.1 Exporting

The call obj2 . export (desc, index) migrates an object obj1, described by desc, along the channel of obj2 indicated by index. Either obj1 . giveSelf () or obj1 . giveCopy () is used to prepare desc. Export uses the migration algorithm of section 6.1. The object on the other end of the channel will receive a special invocation message, signalling the arrival of an exported object.

6.2.2 Importing

The internal interface for requesting an import is sosImport. For C++ programmers, an easier-to-use interface is implemented by a compiler extension: the clause new dynamic (provider) class (importReq, ...) generates a call to sosImport followed by a call to the re-initializer. The arguments are: provider, the reference of an object which will be requested to provide a proxy, and importReq, an import request message carrying untyped request parameters. The AS adds to the import request the reference of the requestor. Possible extra arguments (indicated by the ellipsis) will be passed to the *re-initialization constructor.*

The other arguments to sosImport are automatically generated by the compiler: key describes the expected type of the imported object; and "class" is the name of the class in the new dynamic declaration, which is used to select a default provider.

The mechanics of importation are the following. The AS proxy of the requestor performs a find based on the provider reference. This yields the location of the provider object (or of one of its fragments if the reference was to a fragmented object). The AS proxy at that location then performs the giveProxy upcall on the provider, with a copy of the import request, carrying sufficient information to identify the requestor.

The provider's giveProxy selects some object M to be migrated, and calls either giveSelf or giveCopy, to prepare a description which it returns; alternatively, it may return an error indication. The object M could be the provider itself, or some other object of its context, or a stored object. In the latter case it must be of the same group. When giveProxy returns, M is migrated to the requestor, according to the algorithm of section 6.1.

At the end of a migration (step 5) a re-initialization procedure is up-called, to allow finalization. A typical use of the re-initializer is to set pointers to meaningful values, or to request more importations.

6.3 Migration of code and prerequisites

We mentioned (step 4 of the migration algorithm of section 6.1) that the code and prerequisites are recursively imported, if not already present, before calling

the re-initializer. The pre-requisites are the environment the migrated object needs in order to function; the object's code is just one kind of prerequisite. These are imported if not already present, in order to avoid waste: this allows two imported objects to share code if they are implemented similarly. The same mechanism supports static linking of the code for proxies, without any loss of functionality.

The giveProxy procedure for class code migrates a copy of the code. Two similarly-implemented objects in the same context share a single code object (because pre-requisites are imported only if not present).

Since a prerequisite is imported according to the same algorithm as other objects, its re-initialization procedure is called in step 5. The re-initializer for a code object is a *dynamic linker and type-checker.* The type is checked against the key argument to sosImport. The linking and type-checking algorithm are language-specific; other languages could be supported simply by implementing a new code class.

The strength of this design is that pre-requisites are elementary objects like any other. Dynamic linking and type-checking are automatic, without being wired in. The drawback is that type-checking is automatic only for the first import of an object of a certain class; type-checking for subsequent imports must be special-cased.

6.4 Assessment

We stress that the upcalls to giveProxy and to the initializer, together with pre-requisistes, implement a very important concept: extending a system-defined mechanism with *programmer-defined semantics.* Arbitrary objects can be migrated, and the semantics of their migration is type-specific, above a single, generic, system-implemented mechanism.

6.5 Calling the re-initializer

The C++ syntax for importation is an extension of the instantiation syntax, and in C++ the re-initializer is in fact a constructor. This is appropriate since their semantics are quite similar.[5]

This raises the issue of whether the reinitializer call should be generated by the compiler, or performed by the system. The former solution permits extra arguments (in addition to the import request); the latter allows the system to know that re-initialization has succeeded. We opted for the compiler solution whenever possible, favoring the confort of C++ programmers. However for

[5] We will not discuss this point in any detail since the language interface is out of the scope of this paper.

exports and for pre-requisite imports, the reinitializer can only be called by the system, hence the dual interface.

The compiler decision was bad for two reasons. First, it imposes to treat exports and pre-requisites differently from imports, which is confusing for the users. Second, and more importantly, an operating system should be independent of a particular language implementation; the system solution should be preferred.

6.6 Export vs. import

Exporting is a more primitive operation than importing. In fact, an import could be modeled as an import request, followed by an export from the provider to the requestor. Initially we refused to have an export primitive, because we were concerned with the protection issues involved, and we didn't know how let the target context make use of the newly-available object. Recently we realized that, for some applications, export is the only natural mechanism: for instance, the Images UIMS is modeled more naturally as a window manager exporting event objects to applications, rather than applications polling the window manager for events.

The export mechanism has been implemented, but the proposed interface is not yet available.

7 Assessment of the prototype

We have already pointed out some positive and negative aspects of SOS. On the positive side: the model of elementary objects is simple and powerful; composite and persistent objects, and dynamic linking and type-checking are built on top of elementary objects; fragmented objects give structure to the universe, by extending the object concept over the net. On the negative side: SOS is not completely independent of C++; there are no persistent groups; groups are not real protection domains; the semantics of migration needs to be cleaned up.

We can mention a few more points.

7.1 Permanent pointers

We provide a library of useful pre-defined types. For instance, "permanent pointers" behave like pointers, except that they remain valid across migration [21]. Their use allows to construct a elementary object composed of many interconnected data segments (we do not currently support pointers between objects). They can be used just like pointers, but must be declared differently.

This imposes a slightly unnatural programming style to C++ programmers, but it's better than banning pointers altogether.

7.2 Protection

We mentioned ealier the fact that entering a proxy does not effectively enter a protection domain. Conversely, the client importing a proxy into his context is unprotected against damage the imported code might do. We have side-stepped this issue, considering that the problem is the same with any library.

7.3 Implementation

Initially, the implementation was intended as a quick-and-dirty, throw-away prototype. The kernel is monolithic and poorly designed. The whole system is too big and fragile to experiment easily.

A limitation is that SOS is prototyped on top of Unix. This has the advantage of providing a good development environment, but the drawback is that we haven't acquired experience with implementing an operating system on the bare machine.

7.4 Performance

The prototype is slow. Starting up the SOS environment takes 40 seconds of wallclock time on an otherwise unloaded, diskless Sun-3/60 with 8 Mb memory. The null application, which exits immediately, sizes 287 kbytes text, 74 kbytes data, and 87 kbytes BSS. It takes approximately 0.3 user and 0.15 system second to execute, and mallocs 445 kbytes before exiting. An application which imports a single Name Service proxy and exits, is the same size, and executes in 0.5 user plus 0.5 system seconds.

The explanation for the code size is that the kernel (37,028 bytes text + 11,820 bytes data + 8,348 bytes BSS = 57,196 bytes) and the dynamic linker (34,532 + 8,180 + 792 bytes = 43,504 bytes) are linked in with application. So are the whole standard C (198,128 bytes total) and C++ (34,376 bytes) libraries, and a few others, in case a dynamically-imported proxy needs them (SunOS 3.4 doesn't have shared libraries). The code for a few comonly-used proxy types is also linked statically to speed their importation, e.g. name service (6,536 bytes) and storage service (8,812 bytes). Thus a total of 383,388 bytes is linked by default with every executable.

The huge malloc size is attributable to the kernel pre-allocating a number of tasks, each with its own 40 kbyte stack, for handling incoming invocations. Finally the general slowness has to do with opening and using sockets, and program size, which causes swapping.

8 Future directions

Despite its limitations, SOS is a positive experience. It is a useful environment for prototyping distributed applications. Although we had implementation problems, our initial concepts have been confirmed, and we have discovered new ideas in the process. Operating system-level support for arbitrary, user-defined, medium-grained, migratory objects can be done and is useful. Our elementary-object model is both simple and powerful.

The Proxy Principle was the focal point of our initial design; with hindsight, we see that the fragmented-object concept is a more general and cleaner expression of the same idea: structuring distributed applications.

We must stress again the importance of the givePproxy and the re-initializer upcalls, and of pre-requisites. These give user-defined semantics to a system mechanism. Thanks to them, SOS is an extremely general system and can support many different object semantics.

The SOS prototype is currently used in our project to implement new distributed object-oriented applications, such as a reliability manager, and an original name service [11]. We are also implementing a "proxy generator" to automate the mechanical aspects of programming proxies, servers, providers, and stub procedures.

Thanks to the accumulated experience, together with Chorus-systèmes (and with support from SEPT) we have designed and implemented COOL, an object support layer in the Chorus-V3 kernel [18]. COOL is simpler and more basic than SOS: it includes only object creation, destruction, and migration, which are implemented directly in the kernel, based on the virtual-memory mechanisms of Chorus. Chorus/COOL runs directly on the bare machine; its interfaces are defined independently of any particular language. By using shared libraries and memory mapping, the size of each program file remains very modest. Persistent objects and contexts are integrated in the COOL object model.

References

[1] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. An overview of the Gothic distributed operating system. Rapport de recherche 504, INRIA, March 1986.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[3] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report R-89-1510, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.

[4] David R. Cheriton. The V-Kernel, a software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The clouds distributed operating systems: Functional description, implementation details and related work. In *The 8th International Conference on Distributed Computer Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[6] D. Decouchant, A. Duda, A. Freyssinet, H. Nguyen Van, M. Riveill, and X. Rousset de Pina. Guide : Un système réparti à objet. In *Actes Convention Unix 89*, pages 297–316, Paris, March 1989. AFUU.

[7] Richard P. Gabriel. The Common Lisp Object System. *AI Expert*, pages 54–65, March 1989.

[8] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *Proceeding and additional papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[10] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[11] Jean-Pierre Le Narzul and Marc Shapiro. Un service de nommage pour un système à objets répartis. In *Actes Convention Unix 89*, pages 73–82, Paris, March 1989. AFUU.

[12] Mesaac Makpangou and Marc Shapiro. The SOS object-oriented communication service. In *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.

[13] Mesaac Mounchili Makpangou. Invocations d'objets distants dans SOS. In Guy Pujolle, editor, *De Nouvelles Architectures pour les Communications*, pages 195–201, Paris (France), October 1988. Eyrolles.

[14] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets : l'exemple de SOS.* PhD thesis, Université Paris VI, Paris (France), February 1989.

[15] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–63, March 1987.

[16] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[18] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Frédéric Herrmann, Michel Gien, Marc Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, December 1988.

[19] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[20] Marc Shapiro. The design of a distributed object-oriented operating system for office applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[21] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *ECOOP'89*, Nottingham (GB), July 1989.

[22] SOR. Programmer's manual for SOS prototype–version 4. Rapport Technique 103, INRIA, Rocquencourt (France), December 1988.

[23] SOR. SOS reference manual for prototype V4. Rapport Technique 108, INRIA, Rocquencourt, June 1989.

[24] Bjarne Stroustrup. *The C++ Programming Language.* Number ISBN 0-201-12078-X. Addison Wesley, 1985.

[25] Bjarne Stroustrup and Jonathan E. Shopiro. A set of C++ classes for co-routine style programming. In *Proceedings and Additional Papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

The Clouds Experience: Building an Object-Based Distributed Operating System*

C. J. Wilkenloh, U. Ramachandran, S. Menon, R. J. LeBlanc, M. Y. A. Khalidi†, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabéu, W. F. Appelbe and M. Ahamad

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

Clouds is a native operating system for a distributed environment. In this paper we give an overview of the main ideas behind *Clouds* as well as some of the reasons that prompted us to design a new *Clouds* kernel. The new kernel, called *Ra*, builds on the experience obtained from the first *Clouds* kernel and provides a general framework for implementing a variety of distributed operating systems. We describe the new kernel in detail and show how *Clouds* can be built from the *Ra* primitives.

Keywords: Distributed Operating Systems, Distributed Computing, Operating Systems.

1 Introduction

Clouds is an ongoing distributed operating system project at Georgia Tech. The *Clouds* system was designed in 1983 [All83], and a first version of the *Clouds* kernel based on that design was started in 1984 and completed in 1986 [Spa86,Pit86,Ken86]. In mid-1987 we started designing the second version of *Clouds* and this kernel was completed in mid-1988 [DLA88,BHK*88, BHK*89]. Currently system services for *Clouds* are being implemented and tested.

1.1 Basic Philosophies

The goals of the *Clouds* project is to develop a set of techiques that provide the following:

- A efficient simple implementation of a distributed operating system.
- The operating system must integrate a number of computers, both compute server and data servers into one operating environment. Integration of special purpose machines such as real-time systems and embedded systems must also be supported.
- The system structuring paradigm is the central theme. This should be clean, elegant, simple to use and feasible.
- The *Clouds* operating system should be a general purpose system, configurable to a variety of special purpose needs.

To attain this end, we have decided upon some basic philosophies which have proven succesful in the project.

- First, we advocate a *minimalist philosophy* towards operating system design. Only those functions of an operating system that must be supported by an operating sytem should be in the operating system. We take this one step further and differentiate between the kernel of the operating system and the operating system itself. We also believe the kernel should be minimal, that is only features that must be in the kernel should be there.
- Second, to support the system architecture, we have chosen a shared memory approach, where memory is structured as persistent objects. This concept not only is in keeping with the object-oriented programming

*This work funded by NSF grant CCR-8619886
†Now at Sun Microsystems

paradigms, but also makes the overall system structure simpler and consistent. In this paradigm, many of the complex operating system functions such as I/O and IPC are no longer necessary, making the system simpler.

1.2 The Goals of *Clouds*

The goal of the *Clouds* project is to develop a distributed operating system that provides the integration, reliability and structure necessary to make distributed computing systems easy to use. *Clouds* is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a local-area network. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency, leading to simple and uniform interfaces for distributed processing.
- Support for various forms of atomicity and data consistency, including transaction processing, and the ability to tolerate failures.
- Portability, extensibility and efficient implementation.

Clouds coalesces a distributed network of computers into an integrated computing environment with the look and feel of a centralized, timesharing system. In addition to the integration, it supports an object-based system structuring paradigm and consistency of the data stored in the system.

The paradigm used for defining and implementing the system structure of the *Clouds* system is an object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. The model has been augmented to support atomicity of computation to provide support for reliable programs [All83,CD89].

The rest of the paper is organized as follows. In section 2 we give an overview of the *Clouds* paradigm. Then in section 3 we present an overview of the *Clouds* project at Georgia Tech. Section 4 presents the *Clouds v.1* kernel. Section 5 presents the reasons behind the *Clouds* redesign and a general overview of the new *Clouds v.2* kernel. In section 6 some details of the implementation of *Ra* are shown, and in section 7 we describe the current state of the implementation of *Clouds* on *Ra*. Section 8 reflects on our experiences with the new kernel and finally, we make some concluding remarks in section 9.

2 The *Clouds* Paradigm

All data, programs, devices and resources in *Clouds* are encapsulated in objects. Objects represent the passive entities in the system. Activity is provided by threads, which execute within objects.

2.1 Objects

A *Clouds* object, at the conceptual level, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a *Clouds* object is persistent and is not tied to any process. A *Clouds* object exists forever and survives system crashes and shutdowns (like a file) unless explicitly deleted. As will be seen in the following description of objects, *Clouds* objects are somewhat "heavyweight" and are suited for storage and execution of large-grained data and programs because invocation and storage of objects bear some non-trivial overhead.

The name of an object, also known as its *capability*, is unique over the entire distributed system and does *not* include the current location of the object (objects may move). The capability-based naming scheme in *Clouds* creates a uniform, flat system name space for objects, and allows the object mobility needed for load balancing and reconfiguration.

An object consists of a named address space and the contents of the address space. Since it does not contain a process, it is completely passive. Hence, unlike objects in some object based systems, a *Clouds* object is not associated with any server process. (The first system to use passive objects, though in a multiprocessor system, was Hydra [WCC*74,WLH81].) The address space of an object is structured. The data in an object is accessible only by the code in the object, and not by any other object. Thus the object has a wall around it which has some entry points through which activity can come in. The code that is accessible through an entry point is known as an *operation* of the object. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the entry points (see the discussion on threads).

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. A complete *Clouds* object can contain user-defined code and data, system-defined code and data that handle synchronization and recovery, a volatile heap for temporary memory allocation, a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object, locks, and capabilities to other objects.

2.2 Threads

The only form of activity in the *Clouds* system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. A thread executes in an object by entering it through one of several entry points; after the execution is complete the thread leaves the object. Several threads can simultaneously enter an object and execute concurrently (or in parallel, if the host machine is a multiprocessor).

Threads can span objects and machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the *Clouds* system as lightweight processes that have a stack space but no data space. A thread that spans machine boundaries can be implemented by several processes, using a remote procedure call mechanism.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a call to an operation of another object. When a thread executes this call, it temporarily leaves the calling object, enters the called object, and commences execution there. The thread returns to the calling object after the execution in the called object terminates. The calls to object entry points are called *object invocations*. Object invocations can be nested.

When a thread executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can return result arguments. That is, object invocations may carry parameters in either direction. These arguments are strictly data; they may not be addresses. Note that names (capabilities) are data. This restriction is necessary as the address spaces of objects are disjoint, and an address is meaningful only in the context of the appropriate object.

Unlike processes in conventional operating systems, a thread often crosses boundaries of virtual address spaces. Visibility within an address space is, however, limited to that address space, thus the thread cannot access any data outside its current address space. Control transfer between address spaces occurs though object invocation and data transfer between address spaces occurs through parameters to object invocation.

2.3 Object/Thread Paradigm

The structure created by a system composed of objects and threads has several interesting properties. First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. (Since the objects exists in a global name space, there is no user-level concept of machine boundaries.) Although local invocations and remote invocations are differentiated by the operating system, this is transparent to the applications and systems programmers.

The storage mechanism used in this object-based environment is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, to create the concept of a permanent virtual address space.

Although files can be implemented using objects (a file is an object with operations such as read, write, seek, and so on), the need for having files disappears in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data spaces of objects, structured appropriately. The need for user-level naming of files transforms to the need for user-level naming of objects. Also, *Clouds* does not provide user-level support for disk I/O. In fact there is no concept of disks or such I/O devices (except user terminals). The system creates the illusion of a large virtual memory space that

is permanent (non-volatile), and thus the need for using peripheral storage from a programmer's point of view, is eliminated.

Many distributed systems are message-based, and hence use messages as the paradigm of choice. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Similar to files, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. Objects provide a shared memory implementation, which can be used (possibly through replication) to provide any kind of shared memory, consistent or not. Objects provide an easy to use abstraction for shared memory, a special case of which ("problem-oriented shared memory") is recommended by Cheriton as a powerful tool for programming distributed systems [Che86]. Shared memory (consistent or not) is seen by many as a better concept than messages for programming distributed systems, e.g. Linda [Gel85].

The system thus looks like a set of *permanent* address spaces which support control flow through them, constituting what we term *object memory*. Activity is provided by threads moving among the population of objects through invocation (figures 1 and 2). The flow of data between objects is supported by parameter passing.

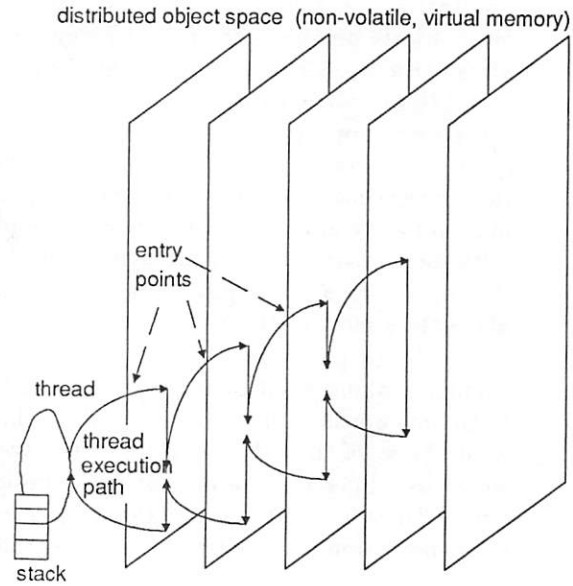


Figure 1: Object Memory in *Clouds*

3 Project Overview

The first version of the *Clouds* kernel has been implemented and is operational. This version is referred as *Clouds v.1* and was used as an experimental testbed by the implementors. This implementation was successful in demonstrating the feasibility of a native object-based operating system, supporting the *Clouds* paradigm. Experience with *Clouds v.1* taught us that the approach works; it also taught us how to better implement the object/thread paradigm.

The lessons learned from this implementation have been used to redesign the kernel and build a new version called *Clouds v.2*. The basic system structuring paradigm used in *Clouds v.1* and *v.2* remains the same. However some of the goals and most of the design and implementation of the system has changed. *Clouds v.1* was targeted to be a testbed for distributed operating system research. *Clouds v.2* is targeted to be a distributed computing platform for research in a wide variety of areas in Computer Science.

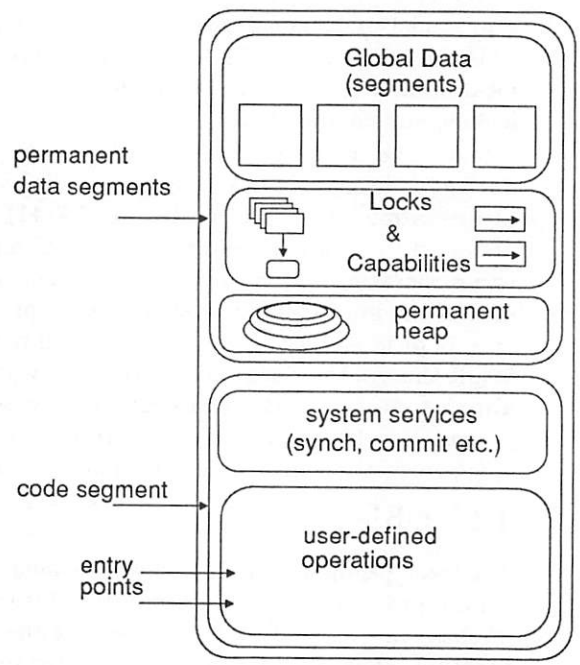


Figure 2: Structure of a *Clouds* Object

The structure of *Clouds v.2* is different from *Clouds v.1*. The operating system consists of a minimal kernel called *Ra*, and a set of system-level objects providing the operating system services. Ra supports a set of basic function of the system: virtual memory management, system object support and low-level scheduling. The system objects provide other systems services (user object management, synchronization, naming, atomicity and so on) and create the operating system environment. Currently the Ra kernel is in operation and the project is involved in implementing most of the system objects.

The basic paradigm discussed above is the common link between *Clouds v.1* and *Clouds v.2*. Both implementations are identical in this regard. Most of the other features of the two are somewhat different. We present our experiences with *Clouds v.1* in brief and the structure and implementation of *Clouds v.2* in more detail.

4 *Clouds v.1*

The first implementation of a kernel for *Clouds* was finished during 1986 and is described in [Spa86,Pit86]. The kernel was monolithic and implemented the passive object–thread paradigm. A kernel–supported extension of the nested action model of Moss [Mos81,All83] made it possible for the programmer to customize synchronization and recovery mechanisms with a set of locking and commit tools.

Since one of the goals of *Clouds* was to produce an efficient and usable system, a direct implementation on a bare machine (VAX-11) was preferred to an implementation on top of an existing operating system such as UNIX. The main goal of the implementation effort was to provide a proof of feasibility of the object–thread model. While this goal was achieved, portability was not a major issue, and as a consequence, it was not easy to port to a different architecture.

4.1 Objects

The basic primitives provided by the *Clouds* kernel are processes and passive objects. The main mechanism provided by the kernel is object invocation. Passive objects in *Clouds* are implemented as follows: the VAX virtual address space is divided into three sections (taking advantage of the division already defined by the architecture). The system section is used to map the kernel code. The process space maps both process stacks (P1 section) and passive object images (in the P0 section). To make the contents of an object visible, the object image must be mapped by the P0 page tables. Furthermore, each passive object can have up to 6 subdivisions of its space, and it is possible to assign different protection attributes to each one of them. This also can be used to share code and data between objects.

4.2 Object Invocation

The basic mechanism used in the *Clouds* kernel to process an object invocation from a thread t executing in object O_1 is the following: t constructs two argument lists, one for transferring arguments to the object being invoked, O_2, and the other to receive the output parameters (results) from the invocation of object O_2 (see [Spa86] for more details). After the construction of the argument lists, thread t enters the kernel through a protected system call or trap. The kernel searches for the object locally and, if found, uses the information in the object descriptor to construct the page mappings for the P0 space. Then, the kernel saves the state of the thread, copies the arguments into the process space (P1) and sets up the new mappings for P0. At this point, the contents of O_2 are accessible through the mappings of the P0 region, and t can proceed with the invocation of O_2's method.

On return from the invocation, the thread t also builds an argument list with the return parameters, and then enters the kernel by means of a protected system call. The kernel now saves the parameter in a temporary area, sets up the P0 mappings for O_1, restores the saved state of t, and copies the return parameters wherever specified by the second argument list constructed by the thread at invocation time.

If on invocation, the kernel cannot find object O_2 locally, it tries to find it remotely. To do so it broadcasts an RPC request. The RPC server in the node that has O_2 acknowledges the invocation request, and creates a local slave process to invoke O_2 on behalf of t. The slave process then proceeds to invoke O_2 locally, and when the invocation completes, it sends the return arguments back to the invoking node. Then the kernel goes on to process the return parameters as described above. This procedure, although seemingly fast (on an Ethernet, only one message is necessary to perform a broadcast), has the disadvantage of making each and every node in the

network perform a search for the object being invoked, substantially increasing the workload on those nodes. The problem becomes worse as increasingly larger networks are considered.

5 *Clouds v.2*

The first *Clouds* kernel served as an existence proof, demonstrating the feasibility of the object/thread approach. However, the first kernel had some problems:

- The kernel was designed with the philosophy that all services provided by the operating system were to be implemented in the kernel. This lead to a large, complex kernel.
- The size, complexity, and interdependencies in the kernel made it difficult to substantially modify the kernel or add functionality without introducing errors.
- Dependencies on the Vax architecture appeared throughout the code making it difficult to port.

In addition, there were other problems such as the overuse of dynamic memory allocation for kernel data structures, and slave reclamation problems in the RPC implementation. While these problems were addressable, the three primary problems would have always remained. Since the *Clouds* system was intended to be a research testbed, this situation was intolerable.

The basic design and philosophy behind *Clouds v.2* is a direct result of the problems we encountered in working with the first kernel. The second kernel is a *minimal* kernel, designed with flexibility, maintenance, and portability in mind.

5.1 The Minimal Kernel Approach

A minimal kernel provides a small set of operations and abstractions that can be effectively used to implement portable operating systems *independently* of the underlying hardware. The rule followed when building such a kernel is: *Any service that can be provided outside the kernel without adversely effecting performance should not be included in the kernel.*

Minimal kernels typically provide memory management, low-level scheduling, and communication primitives such as message-passing or object-invocation. The minimal kernel idea is similar to the RISC approach used by computer architects and has been effectively used to build message-based operating systems such as V [CZ83], Accent [RR81], and Amoeba [TM81].

There are several advantages to the use of a minimal kernel. The kernel is small, hence easy to build, debug, and maintain. Minimal kernels assist in the separation of mechanisms from policy which is critical in achieving operating systems flexibility and modularity [WCC*74]. A minimal kernel provides the mechanisms, and the services above the kernel implement policy. In minimal kernels, most operating system services are implemented above the kernel; these services can often be added, removed or replaced without the need for recompiling the kernel or rebooting the system. The ease of installing or removing system services makes it feasible for the same operating system to support a variety of services or policies which is particularly attractive in a system which is to serve as a testbed for research. Thus, higher-level algorithms can be evaluated, and in some situations tested side by side, or replaced without affecting the implementation of the mechanisms and vica-versa.

5.2 The *Ra* Kernel

The basic, minimal kernel is called the *Ra* kernel. The principal objectives in *Ra*'s design were:

- *Ra* should be a small closed kernel.
- *Ra* should be easily extensible.
- In particular, one of the possible extensions of *Ra* should be *Clouds*.
- It should be possible to effect an efficient implementation on a variety of architectures.
- Robustness and ease of comprehension are also desirable features.

In addition to the above, the implementation of *Ra* should clearly identify and separate the parts depending on the architecture for which the implementation is being targeted. This will then minimize the effort required to port the kernel to different architectures.

As was shown in the previous section, object invocation in the first version of *Clouds* was achieved by manipulating the virtual memory mappings. Thus, it was thought that the provision of sufficiently powerful virtual memory

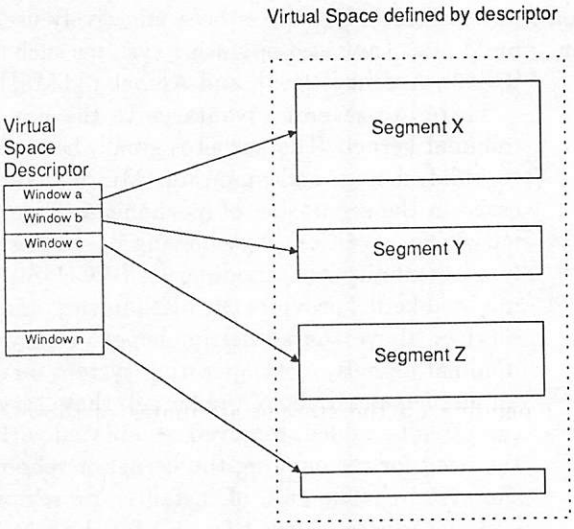


Figure 3: Segments Composing a Virtual Space

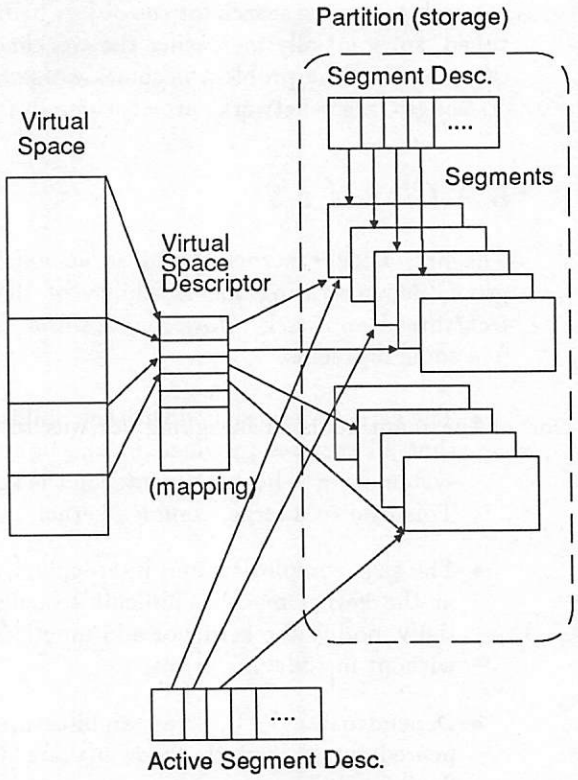


Figure 4: The *Ra* Virtual Machine

manipulation primitives was necessary to implement *Clouds* as an extension of *Ra*. Also, out of our experience with the first *Clouds* kernel, we identified generalizations in the virtual memory management mechanisms which would provide a larger degree of flexibility for the design and implementation of new systems. As a consequence, the following primitives are provided by the *Ra* kernel:

Segments

A segment is a contiguous block of memory. Segments are explicitly created and persist until destroyed. Each segment has a unique system–wide sysname, and a collection of storage attributes.

Virtual Spaces

A virtual space abstracts a complete address space. As such, it specifies how ranges of virtual addresses are to be mapped to ranges of bytes in segments. Each range mapping specification is referred to as a *window*. The kernel interprets the specification represented by a Virtual Space and realizes it on the hardware virtual address space. Such a realization is referred to as a *mapping* of the virtual space. A virtual space layout is described by a *virtual space descriptor* (figures 3 and 4), that is in turn stored in a segment.

Isibas

An *isiba* [1] is an abstraction of activity, and can be thought of as a lightweight process. An isiba can be used as a daemon within the kernel, or with a virtual space to implement 'heavier' forms of activity.

In addition to the above, the kernel assumes the existence of entities, called *partitions*, which store the segments. Partitions are implemented as *system objects* (see next section) and their interface with the kernel is well defined.

Ra also assumes the existence of (at least) three different regions in the virtual address space of the CPU. These regions are referred to as the O, P, and K spaces. To realize the virtual memory specification represented by a Virtual Space, the virtual space has to be *mapped* onto one of the above regions. With the exception of the K space, different virtual spaces will be constantly mapped and unmapped into the P and O spaces. The K space can map only one virtual space: the *kernel virtual space*. Such virtual space will contain the

[1] The term Isiba comes from early Egyptian and means "the light soul of Ra".

kernel plus any other system objects extending the functionality of the basic kernel.

Ra is most closely related to the Mach kernel [ABB*86] by its set of primitives. The Choices kernel [CJR87] also is closely related. During the design of *Ra* we were influenced by ideas from Multics [Org72], Hydra [WCC*74], and Accent [RR81], as have other distributed operating systems designs in the tradition of *Clouds* [NKK86,Nor87].

5.2.1 Extensibility in *Ra*

One of the main goals in designing *Ra* was to facilitate its extensibility. One of the ways in which *Ra* achieves this goal is by implementing only mechanism, leaving the policy of their use outside the kernel. The question still remains, however, of how to specify policies and other extensions to the basic kernel.

To this end, *Ra* provides an interface to outside modules, which we refer to as *system objects*. System objects are used to encapsulate necessary and/or useful operating system services and resource managers that have direct access to the kernel. The system objects in *Ra* are organized in a hierarchy of classes, ultimately deriving from the *SysObj* class. The system objects can be thought of as plug in software modules that can be either linked in with the kernel or loaded dynamically into a running system.

The kernel interface with the system objects is viewed from the system objects as a collection of kernel classes. The kernel classes are collections of kernel data and procedures to access and manipulate that data. The six kernel classes *Ra* provides access to are: the Segment Class, Virtual Space Class, Isiba Class, Synchronization Class, Device Class and System Object Class. The first three support *Ra*'s primitive abstractions. The fourth exports synchronization primitives to maintain the consistency of the kernel data structures and system objects. The fifth class provides a means of installing and removing interrupt handlers and initialization code. The last class provides support for invocation of system objects through a kernel system object directory.

Some system objects are needed for the normal operation of the system, and the kernel assumes their interface incorporates a minimal set of methods that the kernel can use. These system objects provide services needed by the kernel or required to increase the functionality of the system (i.e. a partition from where to obtain more system objects). As an example, *Ra* assumes that a system object implementing a partition has at least the methods *ActivateSegment, DeactivateSegment, ReadPage*, and *WritePage*, although a particular implementation of a partition system object can provide more methods, which can be used by other system objects having knowledge of the additional methods in the interface. Below we discuss two classes of essential system objects.

Virtual Memory Managers.
: The mapping of pages for different types of segments has to be done differently depending on the storage attributes of those segments. Virtual memory managers are then responsible for performing any type–dependent processing on accessing the memory of a segment. Each segment being used by a system has one such manager associated with it, and the particular system object can vary from node to node, even though the segment may have the same storage type.

Partitions
: A partition is responsible for maintaining and manipulating segments. Each segment is maintained by exactly one partition, and the segment is said to *reside* in that partition. The partition in which the segment resides is called the *controlling* partition. Creation and deletion of segments is performed through their controlling partitions. The partition is responsible for maintaining the block tables which describe the segment in secondary storage. Thus, to either read one of the segment pages from or write it to secondary storage, its controlling partition has to be invoked. The partition is notified that one of its segments will be subject to further activity by activating the segment. Similarly, the partition is told that a segment will not be used in the near future by deactivating the segment.

  The storage made visible by a partition at a certain node does not have to be part of one of the node's local disks. It can be storage managed by one or more other partitions at remote nodes. When a segment is activated through such a partition, it is said that it is *remotely activated*, and the partition is referred to as the *mediating partition*. A remote activation requires that the remote segment be first located (that is, it is necessary to find the remote controlling partition

for the segment). Notice that there is no difference between locally and remotely activated segments outside the partition system object. This gives a view of a global segment space shared by all nodes in the system.

When sharing a segment it is necessary to follow some protocol which enforces coherency constraints on the segment. To this end, distributed shared memory is implemented by means of a special mediating partition, called the DSM partition. The DSM partition at a given node communicates with other DSM partitions (or local disk partitions), cooperating with them to implement the coherency constrains for each segment.

6 Implementation of *Ra*

Ra is implemented in C++ [Str86] on a SUN-3/60 architecture. C++ was selected over C due to the extra support for software engineering. The type-checking and class facilities of C++ such as private data and methods, derived classes, and virtual functions make it easier to write modular code and enforce adherence to any defined interfaces. The optional parameter feature of C++ also enables those interfaces to be easily extended while retaining compatibility with existing code.

The design and implementation of Ra is designed to identify and isolate machine dependencies. The code is thus divided into two collections of programs: one which can be ported between machines without change and the other which has to be modified when porting to a new architecture. The machine dependent portions interact with the machine independent portions through a well-defined set of interfaces. To port to a new architecture, the machine dependent portions have to be rewritten, maintaining the set of interfaces expected by the machine independent portions.

The kernel is divided into two major sections: virtual memory handling and Isiba and synchronization handling. Within the virtual memory handling section, three main classes were defined: the *Virtual Space Descriptor*, the *Window* class, and the *Segment Descriptor* class. Each of the main classes defined, provided the procedural interface to be used by system objects using the kernel mechanisms. Virtual memory handling is the main mechanism provided by the kernel, and the data structures used to implement it are described below.

Before any part of a segment can be accessed, the part has to be mapped by one of the windows in a virtual space, and the virtual space has to be mapped into one of the O, P or K spaces. Once a segment has been accessed, it is necessary to keep track of which pages of the segment are being read or modified. Thus, the kernel keeps a structure called the *Segment Descriptor* for each segment which is being accessed. A segment for which the kernel allocates such a data structure is said to be activated. In the current implementation there is a table of active segment descriptors, thus the number of segments which can be active at any given time is limited by the size of the table.

A special type of segments are those containing a virtual space descriptor. The current implementation takes a lazy approach to mapping a virtual space into either the O or P spaces: It does not build any virtual memory mapping until it is needed (i.e., a page fault occurs). Thus, in order to map a virtual space into either O or P space, the virtual space descriptor does not need to be accessed. However, when an address in the virtual space is referenced for the first time, the kernel will have to resolve it. To do so, it first finds to which segment that address maps. Such information is only contained in the Virtual Space Descriptor. Thus, the kernel needs to access the virtual space descriptor in those cases. To do so, it maps the segment containing the Virtual Space Descriptor into a region of the kernel virtual space. In the current implementation, there are 16 regions in the kernel virtual space reserved for mapping virtual space descriptors. Once a virtual space is being accessed, however, the kernel has to keep information about it to be able to map and unmap it from the P or O spaces. This information is kept in a structure referred to as the *Active Virtual Space Descriptor*. A Virtual Space for which the kernel allocates an active virtual space descriptor is said to be activated. Similarly to what is done with the active segment descriptors, there is a table with a limited number of active virtual space descriptors.

A virtual space descriptor consists of a table of *Window* structures. When a mapping is established between a range of addresses in the virtual space and a portion of a segment, one of the window structures is used to record the particulars of the mapping. The task of correctly maintaining the window structures is independent of the underlying machine architecture. However, there is one aspect of the virtual space handling which is

very much dependent on the architecture: physically mapping the virtual space into the addresses corresponding to the O or P spaces.

The *Machine Dependent Virtual Space Descriptor* class isolates these machines dependencies. This class encapsulates the additional data and algorithms needed to adapt the general mechanisms of the Virtual Space Descriptor class to a particular architecture. For the example under consideration, the machine dependent operations implement the particular manner in which the virtual memory mappings are set up, recorded and modified in a particular architecture. The Virtual Space Descriptor Class is defined to include an instance of the Machine Dependent Virtual Space Descripor class. Whenever the machine independent portion needs to perform service that is machine-dependent, it calls upon its machine dependent member to perform the service on its behalf.

The kernel also defines a set of synchronization primitives (semaphores, events, and spin locks) to be used to synchronize isibas executing system code. One of our intentions was to provide an implementation which could be easily adapted to a multiprocessor architecture. Thus, in this implementation, care was taken to control access to the structures which would be shared by the different processors by means of appropriate synchronization primitives.

Some system objects can be understood as providing services for the system in which they are loaded. However, those services will be provided by different system objects in different systems (or the system object implementing a given service can be substituted at any point in time). Thus, it is not advisable to refer to those services with the system names of the system objects implementing them. The kernel defines a directory of system object services. The directory can be accessed by any system object, and its main function is to serve as a basic naming service which identifies which system object (if any) provides a given standard function in the system (for instance the Distributed Shared Memory partition, the system object loader, or the virtual memory manager for a certain type of segments).

The implementation of the *Ra* kernel consists of about 1000 lines of assembly code and 12000 lines of C++ code. Approximately 6000 lines are machine dependent code while the rest are machine independent.

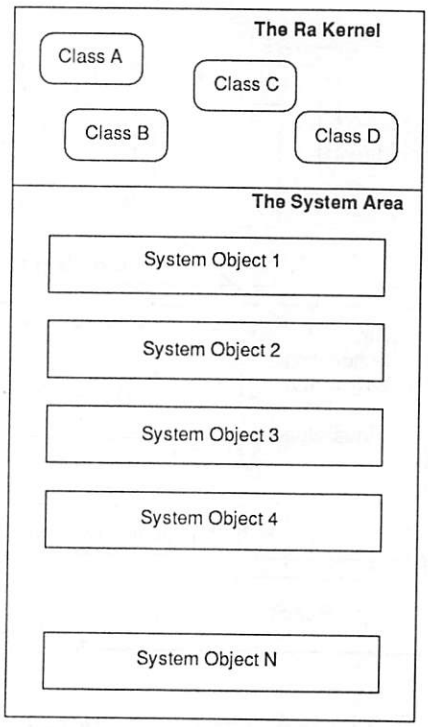


Figure 5: System Objects and System Space

7 *Clouds v.2* and *Ra*

Clouds v.2 consists of the *Ra* kernel plus a collection of system objects implementing *Clouds* semantics (figures 5 and 6). An object is implemented in *Ra* by using a virtual space. The storage of the object is ultimately realized by the segments mapped to by the windows of the virtual space. The *sysname* part of the object capability is the *sysname* of a segment containing the information necessary for the object manager and the *Ra* kernel to properly set up and map the object's virtual space into the machine's O space.

Processes are implemented by associating an isiba with a virtual space which controls per-process memory such as the process stack and the parameter passing areas. Although the virtual space may map many segments into the process's virtual space (which may change over the lifetime of the process), the state of a process may be saved into one controlling segment. This ability to *freeze* a process into one segment and remotely activate that segment (and all necessary segments after that) using DSM will provide *Clouds v.2* with a simple, easy way of performing process migration.

The object invocation mechanism is imple-

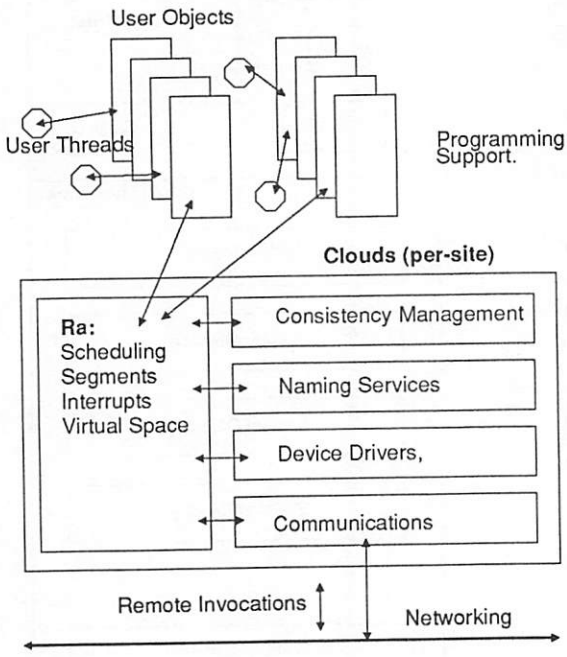


Figure 6: The *Clouds/Ra* Environment

mented by means of a system object: the *Object Manager*. Object invocation then occurs along the lines described for the first implementation of *Clouds*. The current implementation differs in what happens when an invoked object is not stored in a local partition. On the first *Clouds* implementation, such situation would result in an RPC being generated. In the current implementation we have provided the system with an implementation of a DSM partition. Instead of generating an RPC for the invocation, the system may choose to execute the object invocation by remotely activating the segment containing the virtual space descriptor for the invoked object.

Activating an object remotely using DSM is not always better than performing an RPC when the object is invoked. If processes on different nodes have the same locality of reference in an object and they all activate the same object remotely using DSM, DSM will thrash, paging the common pages back and forth between the different nodes. In that case, it might be better to have moved all the computation to the same node by performing a standard RPC where they can physically share the memory.

In addition, if the load on the nodes of the system is not balanced, it may be a better idea to send the computation to a remote node. Notice that the remote node chosen would not have to have the object being invoked, as it could use DSM to access the object's segments. This provides another mechanism besides process migration to balance the load of the network.

The DSM partition communicates with both a local disk partition (if any) and a DSM controller (DSMC), which assists in mapping remote segments (see [RK88] for more details on the implementation of DSM). DSMC has been implemented on Unix as a software module consisting of approximately 3500 lines of C++ and is currently being ported to Ra.

7.1 Implementation Work

When the *Ra* kernel came online, the kernel consisted of the basic kernel plus a few essential system objects including a virtual memory manager, and a partition manager that manages a ramdisk, totalling 13000 lines of code altogether. Since that time, over 11000 lines of C++ code have been added to the system in the form of new system objects, new utility classes such as devices, as well as a lightweight, reliable transport protocol ([Wil89]).

7.1.1 System Objects

The new system objects include a tty driver, an ethernet driver, user object controller, user process controller, system monitor, and unix partition.

Both the tty driver and ethernet driver are *devices* which are themselves *system objects*. Devices in Ra conform to a standard interface: a class definition which is to be used by the device driver writers. Device driver system objects should provide at least a subset of the methods defined in the base class. They are *open()*, *close()*, *read()*, *write()*, *getmsg()*, *putmsg()*, *poll()* and *ioctl()*. The *getmsg()* and *putmsg()* methods are very efficient implementations of *read()* and *write()*, that use Ra-style buffers to transport data instead of an arbitrary length data buffer.

User process controllers must also conform to a standard interface which includes routines to create, kill, sleep, suspend, resume, swap in/out, and freeze/unfreeze processes. All process controllers must support at least a subset of the indicated methods. The current user process controller supports process creation, killing, object invocation and object return, and interacts with the object controller system object to ensure that

Context switch	0.325 msec
Ethernet packet round-trip	3.2 msec
RaTP reliable round-trip msg	7.8 msec
RaTP 8K reliable data fetch	21.6 msec
Segment activation (Unix partition)	22.1 msec

Table 1: Performance of *Clouds v.2*

needed objects are ready to be invoked.

The system monitor provides a low-level monitor/shell capability. The monitor can be used to read and alter values of variables in the kernel or system objects, as well as execute arbitrary methods in the kernel or in a system object.

The Unix partition provides the system with a partition that manages a networked disk. Ra segments managed by the Unix partition seem to the system to reside on the *Clouds* node but actually reside on a Unix system on the local ethernet as Unix files. Reads, writes, and control messages are shipped to the a Unix system where a server operates on the Unix files corresponding to the indicated Ra segments.

7.2 Ra Transport Protocol

The Ra Transport Protocol (RaTP) provides reliable message transactions over the ethernet. This protocol is designed to be connectionless and is efficient for providing the request-reply form of communication that is common with client-server interactions. Since Clouds supports object invocations using RPC or DSM, this is the type of communication that is encountered in the system.

The RaTP protocol has been implemented both on Ra as well as on Unix. In addition to message transaction, the RaTP implementation provides interfaces to the RPC and DSM mechanisms as they are the heaviest users of RaTP. We are currently using RaTP to run the DSM clients on Ra and DSM servers on Unix file servers.

7.3 Performance

Performance measurements for *Clouds v.2* are shown in table 1. All times shown are in milliseconds. The RaTP times are the times to send 24-byte control messages or fetch 8K of data using RaTP from one kernel application to another kernel application. The kernel applications reside on different machines and are kernel-level services (kernel or system objects).

7.4 Work in Progress

There are currently several efforts in progress. At the high level, work is being done in the area of reliability, the thrust of the original Clouds system. We have developed a set of flexible mechanisms that support customized data consistency. In fault tolerance research we have developed a scheme that replicated data as well as computation to guarantee forward progress of computations. Work is underway to design schemes that exploit multicast communication to make a variety of services (e.g. object location, group communication, commit protocols, replication management and so on) more efficient ([AB89], [BA89]).

At the operating system level, we are designing and building the various operating system services and environments. These projects include presistent memory management, object programming support, naming schemes, location services, ASCII I/O services, resource allocators and so on.

At the implementation level, efforts to implement threads, RPC, and DSM are underway. Both the RPC and DSM implementations will use the RaTP communications system.

8 Reflections

In moving to the second version, we made four major decisions:

- To use a minimal kernel approach and provide the system object interface for implementing operating system services.
- To write the kernel in C++.
- To separate mechanisms from policies
- To attempt to isolate machine dependencies.
- To make the operating sytem encompass the minimalist philosophy. Most services will be provided by OS objects that are similar to user objects.

The first two decisions have without question paid off. While the kernel internals are quite intricate, well-defined interfaces exists for requesting kernel services and C++ prevents system implementors from by-passing those interfaces. This makes it possible to hide the implementation specifics of one part of the kernel from the rest of the kernel. This in turn reduces hidden

interdepedencies which makes it easer to change parts of the kernel without breaking the rest of it in the process.

Likewise, the system object interface provides a simple, well-defined interface by which system service implementors can add to the system without having to learn the details of the kernel internals. They need learn about only what they must use and provided they use the supplied interfaces correctly, they can depend on them (by and large) to function correctly. The existence of a simple, uncomplicated method of adding functionality to the system shortens the "learning curve" of new students and makes it possible for them to actively contribute to the project more quickly.

The separation of mechanism from policy is definitely a good idea and is essential for a testbed that is intended to support both mechanism and policy research. While we have not yet been able to completely verify how well we have succeeded in this goal, judging from our experiences in extending the basic kernel, we feel optimistic about the situation.

As far as isolating machine dependencies and the ease of portability, that question can not be adequately answered until we actually try and port the system. However, we are hopeful as we do feel that we are in a better position with the second version, portability-wise, than we were with the first.

We have become comfortable with the minimalist kernel approach. The operating system development process has been sped up by not having to deal with a long list of functions before having a working system. Adding modules is simple, the structure is intuitive and the interfaces are cleaner than in most operating systems.

9 Concluding Remarks

Clouds is intended to serve as a base for research in distributed computing at Georgia Tech, and the new *Clouds* kernel, *Ra*, provides mechanisms for extending its functionality, thus giving extra freedom for the design of systems like *Clouds*. The design of *Ra* benefited from the experience gained from the design and implementation of the first *Clouds* kernel. The additional freedom in *Ra* makes it possible to easily test the designs of a larger variety of systems using *Ra* as the implementation base. This would not have been possible (or easily done) with the original *Clouds* kernel design.

References

[AB89] M. Ahamad and N. E. Belkeir. Using Multicast Communication for Dynamic Load Balancing in Local Area Networks. In *14th Annual Conf. on Local Computer networks*, page , October 1989.

[ABB*86] Mike Accetta, Roberta Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings Summer Usenix*, July 1986.

[All83] J. E. Allchin. *An Architecture for Reliable Decentralized Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983. (Available as Technical Report GIT-ICS-83/23).

[BA89] N. Belkeir and M. Ahamad. Low Cost Algorithms for Message Delivery in Dynamic Multicast Groups. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[BHK*88] José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. Clouds — A Distributed, Object-based Operating System: Architecture and Kernel Implementation. In *European UNIX systems User Group Autumn Conference*, EUUG, October 1988.

[BHK*89] José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. The Architecture of *Ra*: a Kernel for *Clouds*. In *Proceedings of the Twenty-Second Annual Hawaii International*

Conference on System Sciences, January 1989.

[CD89] Raymond C. Chen and Partha Dasgupta. Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[Che86] D. R. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems Design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190–197, May 1986.

[CJR87] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchichal Open Interface for Custom Embedded Systems). *Operating System Review*, 21(3), July 1987.

[CZ83] D. Cheriton and W. Zwaenpoel. The Distributed V Kernel and its performance for diskless workstations. In *Proceedings Of the 9th ACM Symposium on Operating System Principles. (Breton Woods, N.H.)*, pages 129–139, ACM, October 10–13 1983.

[DLA88] Partha Dasgupta, Richard L. Leblanc, and William F. Appelbe. The Clouds Distributed Operating System: a Functional Description, Related Work and Implementation Details. In *Proceedings of the* 8th *International Conference on Distributed Computing Systems*, San Jose, June 1988.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, January 1985.

[Ken86] Greg Kenley. *An Action Management System for a Distributed Operating System.* Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[Mos81] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[NKK86] E. Nett, J. Kaiser, and R. Kroger. Providing Recoverability in a Transaction Oriented Distributed Operating System. In *6th International Conference on Distributed Computing Systems*, pages 590–597, May 1986.

[Nor87] J. Duane Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems.* Volume 16 of *Perspectives in Computing*, Academic Press, 1987. Editors: W. Rheinboldt and D. Siewiorek.

[Org72] Elliot E. Organick. *The Multics System.* MIT Press, 1972.

[Pit86] David Pitts. *A Storage Management System for a Reliable Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[RK88] Umakishore Ramachandran and M. Yousef Amin Khalidi. *An Implementation of Distributed Shared Memory.* Technical Report GIT-ICS-88/50, School of Information and Computer Science, Georgia Institute of Technology, December 1988.

[RR81] R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Spa86] E. H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Technical Report GIT-ICS-86/16).

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[TM81] A. S. Tanenbaum and S. J. Mullender. An Overview of the Amoeba Distributed Operating System. *Operating System Review*, 13(3):51–64, July 1981.

[WCC*74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[Wil89] Christopher J. Wilkenloh. *Design of a Reliable Message Transaction Protocol.* Master's thesis, git-ics, 1989.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp, An Experimental Computer System.* McGraw-Hill, Inc., 1981.

Experiences with Efficient Interprocess Communication in DUNE

Marc F. Pucci *J. L. Alberi*

Bellcore
445 South Street
Morristown, NJ 07960

ABSTRACT

Efficient interprocess communication (IPC) in distributed systems has always been difficult because the underlying network protocols have a great effect on system performance. This problem is compounded if the distributed system must work over several forms of interconnect since designing an efficient solution for one may make accommodating another awkward or impractical. This paper describes a model of interprocess communications that addresses this problem and is generally efficient for many types of network, even nontraditional ones. From our experience in building and using the DUNE distributed operating system, we draw some conclusions about how to structure IPC in distributed systems and what IPC features to avoid.

INTRODUCTION

Distributed operating systems running on native hardware often rely on special purpose protocols for interprocess communication [1][2][3]*. Special protocols provide efficient operation by reducing the layering inherent in general purpose protocols, but are difficult to adapt to new network technology. Systems that require several special purpose protocols to support different communications media are difficult to implement and maintain because of the complexity of the code. General-purpose layered protocols [4][5] provide easy adaptation to new technology and support for multiple networks, but are slow, especially when each layer is mapped into a process [6]. In the DUNE distributed operating system, we have established a flexible yet efficient model that allows incorporation of virtually any protocol to support interprocess communication. This paper describes this model and draws some conclusions about the efficiency of various forms of protocols that support the model.

The interprocess communication model used in DUNE is termed a *service request*, which extends the *remote procedure call* (RPC) [7][8] model of interprocess communication. The RPC mechanism is modeled on the standard procedure call found in almost all high level languages. Data and results are transmitted across a network at the beginning and end of the RPC transaction, respectively. The extensions we have provided address the following points.

First, we are interested in providing complicated services, where the amount, source, and destination of the data passing between the client and the server are unknown in advance. Thus the RPC mechanism must reach back across the network for additional data, as necessary.

---

* Other distributed systems that are built on top of general purpose operating systems like UNIX are not addressed in this paper. Such systems generally establish interprocess communication over the network facilities that the operating system provides and consequently do not permit the tailoring of communications for efficiency.

Second, we want to optimize the transfer of data on a communications channel basis. Breaking up large transfers into smaller packets may be unnecessary for some communications technologies. We want to retain the flexibility of allowing each interface to decide how it should handle the data. For example, if two processors can share a part of their address spaces, physically transferring data with a request for service is unnecessary; the data can be obtained directly when required for almost zero cost.

Third, with limited memory on individual processors the operating system may not have enough space to buffer large amounts of data accompanying a request. An example of this occurs during process migration in DUNE where an entire virtual address space is transmitted. The actual data movement does not occur until the space for the remote process exists. Again, we require a mechanism where the data could be acquired as needed after sufficient kernel buffer space is allocated.

As in the RPC mechanism, each service request appears as a local procedure call to the invoker. The interface to each call includes a binding to the remote part of the request, outbound arguments (which can include arbitrarily indirect references to additional user data), a place to receive results, and an optional *hint* to assist the underlying communications medium in optimizing data transfer [9]. On the remote processor the provider of the service has direct access to the explicit arguments and results of the call, as well as complete access to any indirect data references contained in the request. All access to the latter passes through the hint mechanism of the communications channel for device-specific optimizations. Each channel interface can also resolve general indirect references that are not satisfied by hinting, providing complete (although possibly less efficient) access to the entire caller's environment.

The layer in the kernel beneath the service request is termed an *access method*, which incorporates a set of protocols suitable for the networks in DUNE. A protocol may contain only error and duplicate detection or may be as complex as TCP/IP or the open systems model. We make no architectural distinction among network types, including the null network. There is a null protocol for service requests to the same processor. All higher level functions, *e.g.* flow control, recovery and connection management, are handled in end-to-end fashion at the level of the service request [10].

THE DUNE DISTRIBUTED OPERATING SYSTEM

DUNE [11][12] is a fully functional distributed operating system with semantics of the UNIX operating system [13]. The architecture we will discuss was instrumental in the smooth evolution of this system from its original intent as a non-shared memory multiprocessor system into a distributed system. DUNE eliminates any perceived processor boundaries by distributing both the file system and processing space across all processors in the system. The system-call interface is enhanced from AT&T System V to include the Berkeley UNIX 4.2 network functions. The file system is singly rooted, hierarchical and independent of process location or user identity. Physical storage for the file system is scattered among the processors comprising the system. User processes are uniquely named throughout the system and may migrate between processors either automatically to balance load or by an explicit system call. Signals and process groups are fully developed in accordance with the semantics of UNIX System V.

UNIX is a registered trademark of AT&T.

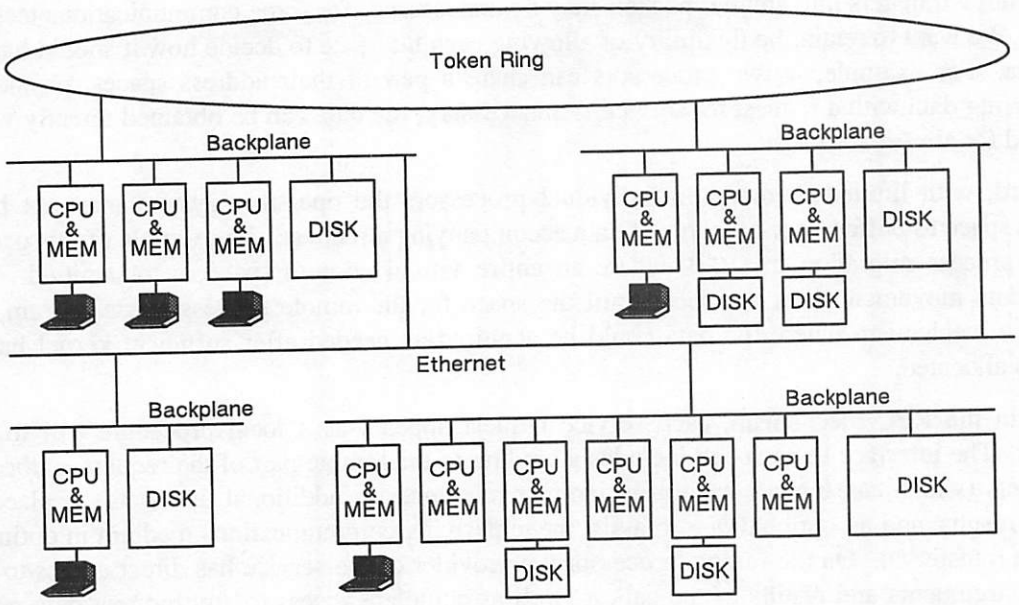


Figure 1. The current DUNE configuration. The Ethernet and token ring are connected in parallel to adapters resident in the backplane, which functions as another form of network.

The current DUNE hardware configuration, shown in Figure 1, simultaneously supports service requests on the same processor, across the same backplane, across an Ethernet™ network, and across a high-speed token ring network. The service request over each type of device is optimized to use specific characteristics of the underlying hardware. The DUNE system is composed of commercially available hardware. The single board computers are based on Motorola 68000 family processors with four megabytes of private memory. Up to eight such processors are connected on a single backplane, which is in turn connected to other backplanes via network interfaces.

SOFTWARE ARCHITECTURE

One of our major design goals in DUNE is efficient interprocess communication. This requirement led us to a structured kernel with network protocols at the lowest levels, followed by access methods, the service request and the higher level kernel functions.

The *service request* is the basic mechanism that distributes work either within the same processor or to other processors. It conceptually transfers the entire client address space to the processor containing the resource to be operated on by the request. This processor invokes local procedures to operate directly (as far as it can tell) on any supplied or implied user data.

An *access method* is a uniform set of functions that supports the service request. An access method includes functions to send hints to the server about incoming data, handle messages, move data between client and server, and send and receive messages. The high level kernel never deals directly with messages but only requests services from resources.

Ethernet is a registered trademark of Xerox Corporation.

The requirements on the network protocol level are minimal. Flow control, connection management, partitioning of data blocks and error processing are all the responsibility of the higher layers. These simple requirements make the complexity of a connection-oriented protocol like TCP/IP redundant, yet DUNE must incorporate widely used protocols.

Although the kernel is well structured, there is no direct mapping of levels in the kernel into processes. For example, the client process executes functions from all levels including, in the case of the token ring, part of the network protocol. The efficiency of DUNE comes from this decoupling of processes and levels within the kernel. In the case of the Ethernet using the TCP/IP protocol, performance is poor, as shown below, partly because the protocol is implemented as a separate process and partly because of the multiple layers in the protocol.

The primary advantage of the layered architecture for general protocols is their uniform interface and modularity. A system or application built using a layered protocol can handle any communications medium with a driver that adheres to the appropriate standard. However, a general purpose standard must accommodate the least capable device it was designed to handle, and imposes constraints that may be inappropriate.

In particular, newer technology provides certain information almost instantaneously – token ring interfaces provide immediate acknowledgement of successful reception of information. Perhaps more serious is the implied breakup of the input data into smaller frames for reliable transmission that occurs in most layered protocols. Although it is possible to treat a common backplane bus as a communications device that adheres to such a standard by breaking large memory transfers into smaller packets, the inefficiency is obvious on this type of network. Data access over a backplane using memory management hardware provides the appearance of transmission without any physical movement of the data.

Uniformity can be obtained by constructing the RPC mechanism around a layered communications system, but we contend that efficiency can be improved by altering the coupling between the remote procedure call and the underlying communications devices. We have created the service request, which encourages specific optimizations for particular devices while providing a simple and consistent interface to the higher level kernel functions. This mechanism is adaptable to nonconventional forms of media; we handle the degenerate case of local communications as well as incorporating shared memory backplane accesses into the communications model.

The Service Request

The service request is different from a traditional remote procedure call because it has, among other features, implicit access to the entire address space of the caller for both reading and writing. Thus complex operations involving unpredictable pointer chasing or address calculations are possible. More details and examples can be found in [11] and [12].

In practice, the entire address space of a client is not needed at the resource. Typically a small but predictable region is sufficient, as long as unpredicted references can be accommodated. The service request permits the calling process to define regions of data that it expects to be used by the resource. As an option, the caller can restrict resource accesses to just the defined regions, thereby providing containment and protection. In general, we have used this feature only for the operating system kernel and not for user processes.

There is a spectrum of information coupling between a client and server. At one extreme, the caller, or caller's agent, knows nothing of the semantic behavior of an operation. Consider an arbitrary UNIX *ioctl* system call – is the third argument a pointer or a flag? At the other extreme, the caller understands the complete details of the operations to be performed, even to the extent of intermediate data structures in use at the server; imagine that the caller understood the layout

of the directory structure of the file system and assisted the server in file *path name* evaluation. DUNE encourages an intermediate level of intimacy: in general, clients or their agents do not understand the details of an operation, but do supply hints to the regions of address space expected to be used by the operation.

For example, a *read* operation clearly defines the destination and size of expected information, and its semantics are independent of the type of file being accessed. It may receive less information than requested, especially for UNIX character-type devices, but this is consistent with the understanding of the hint. On the other hand, a UNIX *ioctl* call can perform any operation on a client's address space. Such operations can also be a function of the kind of device referenced by the call – the same arguments can have different effects on different devices.

Keeping the client's agent knowledgeable of the detailed semantics of all such operations requires too close a coupling between user code (or local system code operating on behalf of the user) and server code at the resource, and is unreasonable to maintain. In practice, we have optimized some common *ioctl*'s dealing with terminals, but have left the majority of such calls unmodified. Relatively mature interfaces, such as *read*, which are clearly defined, are understood in the caller's agent.

Implementation of Service Requests

Our justification for adding complexity to the RPC mechanism is that it permits the lower level communications media to optimize certain data transfer operations. The hint mechanism provides enough additional information to make such optimization possible, while not overly complicating the calling program. Since hints are optional, optimization can be added at a later time.

The basic structure of the service request appears in the caller's agent as:

request (service, resource_link, arguments, results, hint)

Service is the desired operation to be performed on the resource identified by the *resource_link*. *Arguments* and *results* are the explicit inbound and outbound parameters for the operation and can be scalar or arbitrarily chained pointer references. *Hint* is the optional information provided by the caller that describes the regions of the caller's address space that are expected to be used by the service request. For example, the hint for the process migration service request describes the type (outbound user data), address (segment origin) and size (segment length) for the text, data and stack regions of the process.

The same virtual addresses described in the request are established in the server. This is crucial for the proper handling of unexpected data references or operations where complete semantics are unknown. Although it is possible to translate the buffer address implied in a *read* request to a local address appropriate to the server, it is not possible to remap flag arguments or arguments that can function as flags or pointers without knowing how these are to be used.

The service request is implemented on the caller and server processors as cooperating pairs of medium-specific functions, which are summarized in Table 1. The behavior for each function is dependent on the particular communications device (if any) responsible for the connection. The *prologue* sets the stage for the subsequent request. For example, in the backplane access method described below, it is here that the memory management mappings of the buffers described by the hint will occur. If any data are to be pre-sent, as in the token ring access method, this will also occur here. The *request* transmits the actual message (if necessary) for queueing the desired operation at the server responsible for the resource.

Function	At Client	At Server
prologue	act upon hint	
		act upon hint
request	transmit request	
		await request, activate server
expected data request	(not involved)	access as though local
unexpected data request	perform desired access	request data from client
response	await results	transmit results
epilogue	tear down hint	
		tear down hint

Table 1. The structure of a service request. Time increases in the downward direction.

During the actual use of client data at the server, a particular datum may or may not have been described by the hint mechanism. An expected datum reference will be satisfied on the server processor without any intervention on the client's processor. An unexpected datum reference will result in a delay as the server requests the datum from the client's processor. The client, which is awaiting a response indicating it may proceed with its local execution, instead receives a request for access to its address space. It performs the access, returns the datum, and awaits the completion response or an additional request for data.

Once the request has been satisfied, the *epilogue* tears down the hint mechanisms established by the prologue. Any memory management mappings are invalidated, and pre-sent data are freed.

A further optimization occurs when the hint describes a small amount of data. Rather than using the prologue and epilogue functions for coordinating the data access, space for the data is allocated within the message used for the request and response. Currently, a hint composed of up to three defined regions and a total of 96 bytes of data can be so accommodated. Under these circumstances, a remote service request consists of simply a request and a response message. The code used by the server to access client data conceals whether the data are attained through the message, the prologue/epilogue, or remote client access.

Local Service Requests

We are extremely concerned with the performance of the IPC mechanism in the degenerate case – when requests are satisfied on the same processor. For the sake of uniformity, we require that local operations continue to use the same IPC interface as truly remote operations – we do not want to litter the system with occurrences of:

```
if ( local )
        optimized code;
else
        use ipc interface;
```

as such usage is clumsy and error prone. Neither can one use early binding to convert general service requests to local function calls at build time because resources can move dynamically between processors, thereby altering the linkage between clients and servers.

In particular, we want to avoid the expense of using messages when simple procedural linkages will suffice. Clearly, the standard protocol layering does not anticipate the kind of short-circuit we desire. Even though the cost of using a communications device will dominate IPC time, we expect to service a great many requests locally and therefore require optimized

Condition	Time (in μs)
Local, fully optimized request	210
Above + formatting and queueing messages	540
Above + separate server process	840
Above + using interrupts for local delivery	1170
Remote backplane request	1200

Table 2. Round trip request times and the costs of layering.

performance in the local case. The following timing measurements justify our concern.

Table 2 illustrates various levels of optimization. Data were obtained by constructing additional access methods with the specified characteristics (e.g., queueing messages, fielding interrupts, etc.) and measuring several thousand iterations of a simple system service (*seek*). The first entry corresponds to complete optimization – no messages are used and no context switches occur. The latter is possible because the request mechanism is synchronous, i.e., a requester suspends execution until the result is obtained. Under these circumstances, it is possible to use the flow of control of the requesting process to satisfy its own service request, and avoid two context switches. These measurements also include user-to-kernel system call overhead.

Formatting, queueing, dequeueing and unformatting messages (one for the request, one for the response) add 330 μs. Most of this time is spent in packaging the parameters that identify the requester (user id, quotas, etc.).

The next entry includes the overhead of using separate server processes (as would be necessary if requests were non-blocking). Each process switch (client to server for request, server to client for response) adds another 150 μs.

Finally, if interrupts are used to deliver local messages, thereby fully mimicking the remote style of IPC, latency for the two interrupts involved adds another 330 μs. This brings the local time in close agreement with the instantaneous communications available via the shared memory (backplane) medium shown in the last entry. The small difference between the backplane time and that for the local case with server and interrupts is attributed to two factors that almost cancel each other. 1) There is greater expense in generating an interprocessor interrupt than an internally generated programmed interrupt. 2) Even though requests are synchronous, there is a small degree of parallelism as a request suspends on the local processor while the service begins on the remote processor.

By fully implementing the traditional IPC delivery requirements, the layered protocol communications interface can be used for both local and remote requests, yielding uniformity but at the expense of efficiency. Alternatively, by modifying the basic structure of the IPC model, the service request gives improved performance, with uniformity.

Backplane Service Requests

The transmission of information across a backplane bus, where multiple processors have direct access to common memory, is another case where the traditional protocol layering does not exploit the capabilities of the hardware mechanism. The backplane is used as a communications device – processes do not share data in an unconstrained manner across processor boundaries. Once again, we require that the IPC mechanism appear uniform to its invoker.

Backplane service requests can be optimized by taking advantage of memory management features of the processors, thereby avoiding any data copying. The private memory of a processor can be selectively mapped onto the address space of the backplane where it can be accessed directly by other processors that share the bus. Under these conditions, data movement is instantaneous and error free.

Unlike local requests, messages are now necessary to package and queue the requests and responses that cross processor boundaries. Backplane messages are allocated from a pool of memory common to all of the processors. Actual message transmission merely links the address of a message onto the receiver's queue – the message is not copied. The receiving processor is notified by a mailbox interrupt that schedules a server to process the request.

The movement of any data associated with a request or response is averted by the memory management mappings. The hint mechanism provides the backplane interface with the location and size of any data regions necessary for the operation. The memory management hardware of the sending and receiving processors can be configured to map a buffer out from the sender's and into the receiver's address space, providing direct access to data for the duration of the request.

Network Service Requests

Interprocess communications over a general network also use the service request to optimize data transfers. DUNE has two networks that support service requests, an Ethernet and an 80 megabit token ring.

The service request model supports minimizing the number of messages sent and received during an operation. If the amount of data is small enough, it is placed in the request message for a write operation or in the response message for a read operation thereby eliminating another message. The situation is more complicated when the data do not fit in the request and must be sent separately.

Data too large to be held in a message are sent to the server before the request. This action implements the access method hint for these two networks. Because the server that receives the write request does not allocate buffer space for the data until it begins processing, the pre-sent data are tagged and temporarily sequestered by the kernel. The tag allows a server to retrieve the information locally and avoid the expense of more messages over the network. The data are retained on the server until the associated service request completes.

Under heavy loads, the server may have no space for the data contained in a hint. The hint is then discarded, and the server must negotiate with the client via the network to retransmit the data at the appropriate time. The server also uses this mechanism when it requires unanticipated data.

Within the architectural framework of the service request, the network hint and shared memory hint are identical even though they are operationally different. The shared memory hint is a mapping function of the processors' memory management units while the network hint requires sending and receiving messages. The local service request bypasses hints entirely. The hint mechanism is generally broad enough to improve the efficiency of almost any network.

The protocol for the 80 megabit token ring is built into the functions that make up its access method. Therefore the protocol layer is distributed among the client process, the server process, and the interrupt code. The hardware handles low level acknowledgements and error detection. Because of the possibility of data errors, the protocol protects against duplicate packets. Higher level protocols, which apply to all access method, are contained in the service request and in kernel functions residing above the service request.

The Ethernet supports TCP/IP in a separate layer beneath the access method functions. The protocol is contained within a separate process that communicates with an intelligent Ethernet interface on one side and the access method functions on the other. Therefore activation of this protocol implies a context switch, which is costly even at the kernel level, and the management of message queues on each side of the process.

PERFORMANCE ANALYSIS

The previous sections have described the IPC architecture for DUNE. The following is a set of measurements under controlled conditions that illustrate the system performance.

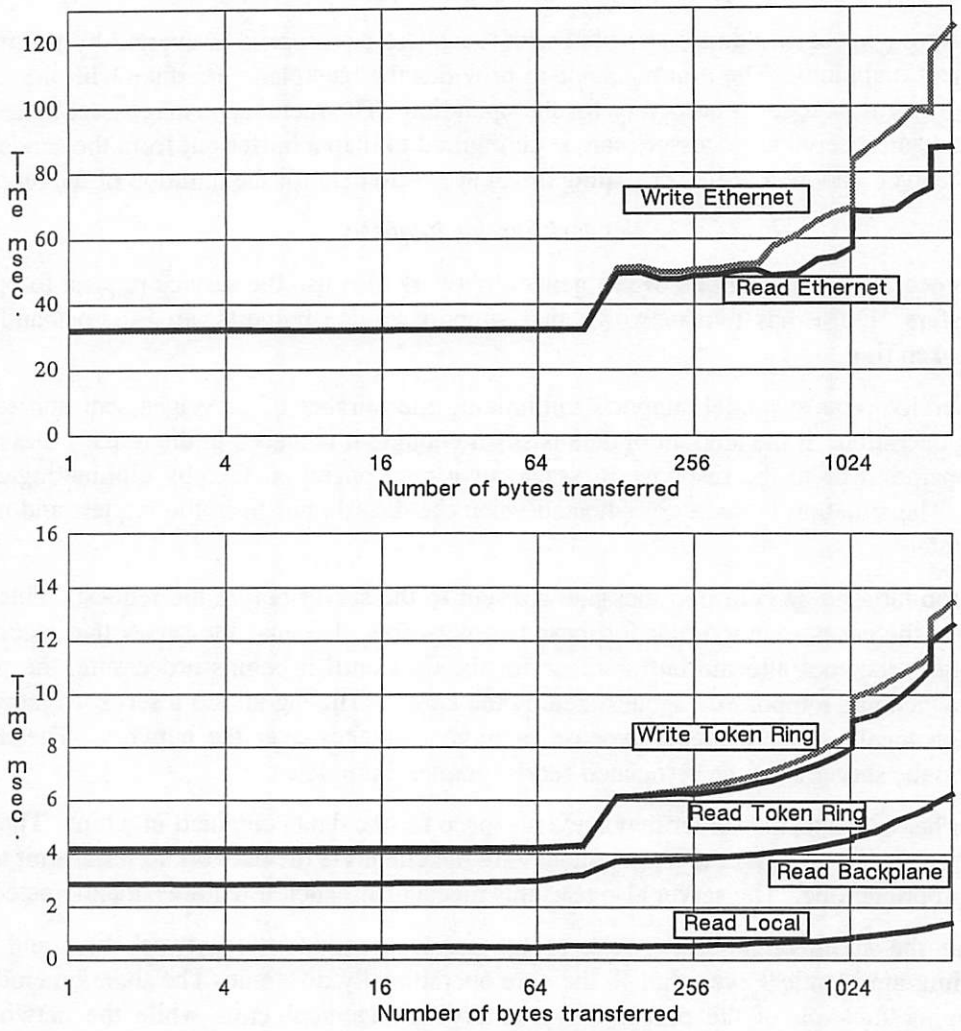


Figure 2. Transfer time as a function of data size for several communication fabrics and optimization levels. The lower graph illustrates (from bottom to top) reading from a local processor, reading across a backplane, reading across a token ring, and writing across a token ring. The upper graph shows reading and writing across the Ethernet using the TCP/IP protocol. Note the change in scale.

A test driver is installed in DUNE for demonstrating the improvements to data transfer through optimized access methods. The test device transfers an arbitrary amount of data between user space and a 1024-byte kernel buffer with the same algorithms and kernel functions as a file transfer. The data are not interpreted, but the data transfer and hint functions are a significant part of any access method and are critical to any measurement of performance.

Measurements depicted in Figure 2 show the time to transfer a block of data versus the amount of data transferred. The abscissa is the number of bytes either read or written, and the ordinate is the amount of time needed to make the transfer. The test driver is the source and destination of the data – data movement from or to a real device is subject to mechanical delays and would mask the performance of the communications networks involved.

Several features evident in Figure 2 are the result of the performance improvements described above. The line labeled ''Read Local'' shows the transfer time for reading through the local access method as measured on a 68020 processor module with a 20 MHz clock. The write time in this case is not plotted as it is similar. The read time for one byte is 0.51 ms, and the time increases linearly thereafter at the rate of 0.38 μs/byte. For comparison, a commercial 68020 system running at the same clock rate† also has a one byte read time of 0.51 ms. Thus the service request with the local access method does not penalize performance in the local case.

The penalty for having a remote server is illustrated in the plot labeled ''Read Backplane''. Here the client code executes on a 68010 with a 10 MHz. clock, and the server is the processor described above. Because the read and write times are similar, only the read time is shown. The single byte transfer time is 2.9 ms. There is a step increase in transfer time at 96 bytes when the data can no longer piggyback on the service request message. The increased processing adds 0.5 ms to the transfer time at this point.

The two plots labeled ''Read Token Ring'' and ''Write Token Ring'' illustrate data transfer via the token ring. The basic 4.2 ms read or write time for a single byte of data reflects the network protocol processing and latency. The increased processing and message time is also evident when the data size reaches 96 bytes, but the equality of the read and write times at this point illustrates the lack of extra messages to fetch the data on writes. The subsequent divergence of read and write times arises when the client copies the pre-sent data to its final destination under program control. The divergence continues smoothly until the discontinuities at 1024 and 2048 bytes where multiple data messages are sent.

The two plots labeled ''Read Ethernet'' and ''Write Ethernet'' have features similar to those of the token ring except for the degraded performance. A single byte of data is written or read in 33 ms, and the break in the data at 96 bytes transferred is also evident. There are also the same discontinuities at 1024 and 2048 bytes, indicating transmission of multiple messages. The minor structure between these discontinuities arises from the fragmentation of messages and data into IP packets within the TCP/IP protocol.

Relationship to Other Work

The service-request paradigm is an extension of the semantics of the remote procedure call mechanism. The changes we have made address our needs to adapt to different underlying fabrics and handle complex operations. We have found it particularly useful to move information between client and server in an unstructured fashion. Supporting complex operations improves performance by reducing the number of interactions across the network that are required for a high-level user request.

While many systems have used RPC-like mechanisms for interprocess communications, Accent [14], Mach [1] and the V kernel [2] have features comparable to ours. Both Mach and Accent use features of the processor's memory management hardware in uniprocessor and multiprocessor systems to pass messages and to avoid the overhead of copying data between processes. The access method for backplane interconnections works in much the same way. Connections for transferring data are made by mapping areas of memory between processes. Partial and random access may then be made to the data. The V kernel, which is a network based system, can attach data to a request message although it is not clear to us that its protocol driven

† A Sun Microsystems Model 140 reading /dev/mem.

mechanism can accommodate unforeseen data transfers.

Mach, Accent and V are all message based systems, using explicit send and receive primitives for communication. While DUNE uses messages within some access methods, they are neither evident to higher levels nor a part of the structure of the system. Significant performance improvements are possible with local operations when the formatting and queueing of messages are avoided.

The access method for the local-area networks can both pre-send data and transfer any unforeseen items in the client process's address space. Mach's *network servers*, implemented as user processes, perform interprocess communication over the network. They correspond closely to the Ethernet access method and, although general and flexible, may suffer from non-optimal performance due to the extra layering and context switches. The Ethernet based RPC described in [7] also bypasses the standard layers for a local network, but uses the protocol hierarchy for internetworking.

We have extracted features from the above systems and encapsulated them into extensible access methods that keep the details of the communication network hidden from the upper layers of the system, while providing increased efficiency.

CONCLUSIONS

The networks supported by DUNE give some insight into the service request architecture as well as the implications of using special and general purpose protocols. The access method for each communications fabric is implemented differently even though functionally they are essentially the same.

The null access method for local service requests preserves the uniformity of the architecture while not compromising the efficiency of the system. This has proven to be quite useful, especially when the automatic load balancing features of DUNE dynamically rebind remote connections into local ones.

The backplane access method demonstrates the flexibility of the service request architecture by efficiently treating a nontraditional interconnect as a communications network.

The protocol for the 80 megabit token ring gains efficiency by eliminating additional processes and collapsing layers within the system. In contrast, the use of a layered protocol for the Ethernet permits it to function on long haul networks but at the cost of decreased performance. Service requests over the Ethernet are measured to be approximately seven times slower than those over the token ring. We attribute this to the complexity of the functionality in the general purpose protocol and the additional processes used in its implementation.

We conclude from our experiences with networks and protocols that coalescing layers and moving end-to-end functionality to a high level directly improve system performance. DUNE demonstrates that a distributed operating system based on these principles can run efficiently on both multiprocessor and distributed hardware by including device specific optimizations for disparate networks.

REFERENCES

[1] M. J. Accetta, *et al.*, "Mach: A new kernel foundation for Unix development," in *Proc. Summer Usenix Conf.*, July 1986.

[2] D. R. Cheriton, "The V kernel: a software base for distributed systems," in *IEEE Software*, Vol. 1, No. 1, 1984.

[3] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proc. Eighth Symp. Operating Systems Principles*, Pacific Grove CA, 1981.

[4] "DOD standard transmission control protocol," in *RFC-761*, Information Sciences Institute, Marina del Ray, CA, January, 1980.

[5] "Transport Protocol Specification," in *ISO/TC 97/SC 16, N 1169*, International Organization for Standardization, June, 1982.

[6] D. D. Clark, "The structuring of systems using upcalls," in *Proc. 10th Symp. on Operating Systems Principles*, December 1985.

[7] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," in *ACM Trans. Comp. Sys.*, Vol. 2, No. 1, February 1984.

[8] A. Z. Spector, "Performing remote operations efficiently on a local computer network," in *Commun. ACM*, Vol. 25, No. 4, April 1982.

[9] D. B. Terry, "Caching hints in distributed systems," in *IEEE Trans. Soft. Eng.*, Vol. SE-13, No. 1, January 1987.

[10] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design" in *Proc. 2nd International Conf. on Distributed Comp. Sys.* April, 1981.

[11] J. L. Alberi and M. F. Pucci, "The DUNE distributed operating system," *Proc. 1988 Using National Conference*, Denver, CO. Also available as a Bellcore Technical Report, 1987.

[12] M. F. Pucci and J. L. Alberi, "Optimized communication in an extended remote procedure call model," *Computer Architecture News*, Sept, 1988. Also available as a Bellcore Technical Report, 1987.

[13] D. Ritchie and K. Thompson, "The UNIX timesharing system," in *Bell System Technical Journal*, vol. 57, no. 6, part 2, July-August 1978.

[14] R. Fitzgerald and R.Rashid, "The integration of virtual memory management and interprocess communication in Accent," in *ACM Trans. on Comp. Sys.*, Vol. 4, No. 2, May, 1986.

Using Transputer Networks to Accelerate Communication Protocols

Horst Schaaser
Hewlett-Packard Laboratories Bristol
Filton Road, Stoke Gifford, Bristol BS12 6QZ, United Kingdom
hs@hplb.hpl.hp.com

Abstract

This paper describes experiments to determine the capability of multi-transputer systems for speeding up layered communication protocols. Some layers of the OSI model (Data Link Layer, Network Layer and a test harness in place of the Service User) have been placed on different transputers.

The performance of these protocol implementations can be best measured by using two identical stations, each consisting of several transputers, and interconnecting the two stations directly.

To measure the user data transfer rate between the two stations the information transfer is kept free from interrupts: Only data transfer packets are processed; call set-up and call-clear packets are not included in the experiments. The system performance is measured for a wide range of message lengths (from 2 bytes up to 4 kBytes). Increasing the number of tranputers per station gives significant speedups for some message lengths.

Another way to increase throughput is task prioritising for processes on the same transputer. To gain a detailed understanding of the underlying phenomena, for example the relative importance of computation and communication, the data flow of protocols is modelled on the transputer system and compared with the performance of the real implementation.

1 Introduction

The speedup of communications software is of considerable interest to various industries. Several multi-processor systems have been built with this specific application in mind; a famous example is the *BBN Butterfly* [Bb85, Fr84] used to execute the protocols for DARPA satellite networks. With the advent of the transputer microprocessor family [I88a, I88b] there has been some interest in using transputer multi-processor systems for accelerating communication software: the CCITT recommended protocol X.25 has been implemented, initially on a single transputer [Bo88], and an interface box to interconnect between primary rate ISDN and Cambridge Fast Ring LAN has been built [Bu88, Cl88, Bu89].

The motivation for using transputer networks stems from two sources: firstly, transputers can be easily interconnected using their serial links. Secondly, protocols in the OSI model of communication [Is84] are organised in a "stack", which may be implemented as a pipelined structure. A layer in the OSI protocol stack communicates only with the layer above or below, where applicable. Thus the stack organisation of the OSI protocol model

maps very well onto transputer networks, and the current limit of four serial duplex links per transputer is not a restriction.

It is worth noting a suggestion by Jensen and Skov [Je88]. They propose to build a pipelined *message passing* multiprocessor system to execute the OSI stack. As they are targeting their system for Gigabit/sec transfer rates they want to use processors of their own design to achieve the necessary high throughput.

The transputer applications mentioned above have been very specific. This paper attempts to describe some more general work about the performance of protocols on multi-transputer systems. The focus is on how to achieve a maximum user communication *throughput* for a variety of hardware and software configurations. There are different ways to achieve high throughput, e.g. using sophisticated protocols or using multi-transputer systems. The experiments described in this paper are about the latter, thus the protocols used are simple.

2 Experimental setup

To obtain experimental information two identical stations have been built, each consisting of a small network of transputers. All the transputers in the system are T800 transputers running at 20 MHz, with all their serial link interfaces switched to a raw data transfer rate of 20 Mbit/second. (Overheads such as control bits in addition to the data, as well as a finite channel setup time [I88a] cause the effective transfer rate to be much lower.) All transputers have sufficient zero-wait-state memory. The software has been developed in the parallel programming language Occam-2 [Po88, I88c].

The two stations were interconnected directly. Call set-up and call-clear are not included in the protocols which permits the experimenter to concentrate on *continuous* information transfer between both stations. A schematic diagram is shown in figure 1. In the experiments, the two stations form subparts of a transputer network. The system is controlled via an interface implemented on a separate transputer, and the "outside world" consists of a PC hosting the transputer network. The interface preprocesses the experimental data produced by the network and sends the results to the host PC.

Both stations transmit and receive at the same time. The processes executed on each station are shown in figure 2. Each station software consists of the data link layer (OSI layer 2), network layer (OSI layer 3) and service user. These three entities constitute the main processes in fig.2. They contain sub-processes, e.g. the network layer contains a multiplexer (MUX) and a de-multiplexer (DE-MUX). Note that the complete user application data (and control bits, where necessary) are passed between the three main processes of a station.

2.1 Service user

The service user in this implementation is a test harness consisting of a sender and a receiver, both executing concurrently. The sender process first generates a message of 4 kbytes in length and stores it. Subsequently, within the service user the following two processes take place concurrently:

- A part of the stored message is taken by the sender and sent out $N+1$ times (N=256) to the network layer.

- The receiver process (on the opposite station) then accepts the first message from its network layer and checks each bit of this message. The successful checking of the first of each $N + 1$ messages triggers the bandwidth measurement by causing the receiver to note down the timer value. The receiver process then accepts N more messages of the same message length from its network layer and checks each bit of the incoming messages. Then the time T which has elapsed since the last timer interrogation is determined. Both T and the message length are stored.

These two processes are repeated for different message lengths between 2 and 4k bytes. The transmission of a large range of message lengths is useful for collecting data for most message lengths used in communication networks, e.g. 128 bytes is the default message length in X.25 networks but other options are possible [Ta81]. Note that the checking in the receiver is not necessary for the correct functioning of the protocol stack. Checking has been chosen as an example for an user application program which consumes CPU time proportional to the length of the received message.

When the measurements are complete, only one of the two stations passes the collected measurement data to the interface.

2.2 Network layer

The network layer is connected to the service user (SU) and the data link layer (DLL). It consists of a multiplexer and a de-multiplexer process acting concurrently. As shown in fig.2, the multiplexer scans 16 channels named "from.SU[i]" (i= 0,1,..15), whilst the de-multiplexer feeds into the channels "to.SU[i]" (i=0,1,..,15). A *single* live service user SU[0] makes the interpretation of the measurement data easier. Therefore only the channels with i=0 are connected to a live service user, all the other channels are dummy channels. Handling the dummy channels, however, creates extra overhead in the network layer.

The multiplexer adds an additional byte to the incoming message to indicate the destination address, and sends the packet out to the channel "to.DLL". The de-multiplexer does the reverse for packets coming from the channel "from.DLL".

2.3 Data link layer

The data link layer consists of a stop-and-wait (SAW) protocol [Ta81] and a buffer which can store a single message. This buffer is essential to avoid a deadlock which would occur if the SAW protocols on both stations wanted to send (or receive) simultaneously. Note that the data link layers in the two stations differ in only one respect: during the start-up phase one of the SAW protocols transmits while the other SAW protocol is in receiving mode. The completely symmetric case is possible as well but shows some slightly different behaviour [Ta81] which is not discussed here.

In this experiment it is assumed that the channels connecting the peer processes are noise-free. Therefore no checksum test is implemented. The remaining tasks for the SAW protocol are:

- Build data packets which include sequence numbers and acknowledgements.
- Transmit data packets sequentially to its peer process and control the flow.
- Check acknowledgements.

- Cope with timeouts.
- Allow retransmission.
- Avoid passing duplicate frames to the network layer.

Essentially, the SAW protocol solves its tasks by sending one message, stopping and waiting for an acknowledgement of the message sent. Upon the arrival of a data packet on which the correct acknowlegement "piggybacked", this cycle repeats. The SAW protocol is a special case of a sliding window protocol [Ta81] which is frequently used, e.g. in the data link layer of X.25. If the sender "window" of a sliding window protocol is not permitted to open for more than one unacknowledged packet it effectively becomes a SAW protocol.

3 Experimental results

3.1 Study of the data link layer performance

It is instructive to study a simplified system first which omits the network layer. Therefore the channels "to.DLL" and "from.DLL" are directly connected to the service user SU[0]. In addition, the checking in the receiver process of the SU[0] is not used. The other functions in the service user are as described in section 2.1. In fig.3 the data transfer rate is plotted against the message length (as transmitted by the sender) and is labelled "real system". (The other curve is discussed further below.) Note the logarithmic scale on the horizontal axis. The software for each station was executed on a single transputer.

The phenomena affecting the data transfer rate are:

1. The SAW protocol's header processsing is independent of the message length. The header processing consists of those operations which are not channel transfers, e.g. the building of new packets, the checking of control bits, etc.
2. After an initial setup phase, the data transfer over physical channels ("hard channels") connecting two transputers is done by the link interfaces [I88a] concurrently with the execution of the CPU in each transputer, using direct memory access (DMA) [At87].
3. For long messages the transfer down a hard channel is proportional to the message length [I88a]. This can produce a large delay for the whole system if the other processes need messages to be transferred down that channel.

One explanation is as follows: For small messages the SAW protocol's header processing dominates while the transfer over hard channels is relatively quick and can be executed in parallel to the SAW protocol. Doubling the message length introduces only a minor delay caused by a longer transfer over "soft" channels between processes on the same transputer. Therefore the overall efficiency increases. For long messages the SAW protocol's header processing becomes negligible and the time for the total system speed approximates the speed over the hard channel. Therefore, in the limit of long messages, the curve in fig.3 approximates a horizontal line.

A confirmation of this hypothesis can be obtained from modelling the *communication part* of the processes. This modelling can be done on the multi-transputer system itself. As before, one processor per station is used. All processes are unchanged, except that the process modelling the SAW protocol consists of a sequence of pure channel transfers to link the sender, receiver and the other station in a similar way as the SAW protocol does.

But no control bits of the incoming/outgoing messages are checked, removed or added, respectively. Thus the header processing of the SAW protocol has been omitted. In fig.3 the curve labelled "channel transfer" is the transfer rate derived by this modelling.

By subtracting the time needed by the model system from that of the real system we can calculate the time not expended on communication (the curve labelled "remainder" in fig.4). The transfer rate needed for unidirectional transfer between transputers is plotted in this figure for comparison. The unidirectional transfer uses the Occam-2 variable array transfer protocol [Po88, I88c] (which is used for almost all channels in fig.2). One can see that the remainder part dominates for small messages and should be a good approximation of the header processing part of the SAW protocol. The curve of the remainder is fairly constant (The independence of SAW protocol's header processing part from the message length has been noted at the beginning of this section). For long message lengths, however, the remainder increases significantly. The nature of this behaviour is being investigated. Fig.4 also contains a plot of the share (dotted curve) which the remainder part has in the total system time. Whilst for small messages almost half of the system time is expended on the remainder part, for large messages this ratio goes smoothly down to about 3%.

To measure the amount of time the SAW protocol is *waiting* for an answer from the other station, one can "shortcircuit" the sending data link layer so that packets are routed back to the same SAW protocol without ever leaving the transputer on which the SAW protocol executes. A plot of the waiting time relative to the execution time of the data link layer is given in fig.4. This plot reveals that a significant amount is spent on waiting. For long packets the waiting is largest, e.g. 74% for messages 4 kbytes long. The wait time can be decreased, however, by replacing the SAW protocol by other protocols which can send more than one message without acknowledgement, e.g. sliding window protocols [Tan81].

3.2 Adding message checking in the service user

This section describes the system behaviour when the receiver process of the service user also *checks* the incoming messages (see section 2.1). The network layer is still omitted. Other aspects discussed are the effect of utilising more processors and the prioritising of some of the processes.

In the two processor station the service user SU[0] now has a processor for itself. Some buffers have been added between the service user and the data link layer to allow for optimal concurrency between DMA supported link transfer and CPU execution. The checking in the receiver consists of comparing the received message with the original message. For long messages the associated effort is proportional to the message length. Consequently, in fig.5 a reduced system performance can be seen especially for long messages (compare with the curve labelled "real system" in fig.3).

If all processes have the same priority two effects can be noted: firstly, the corresponding curves in fig.5 are not smooth. Secondly, there are large differences in performance between the one and two processor stations. To explain this behaviour one has to has to consider the effect of the micro-coded scheduler, which is implemented on each transputer for pre-emptive multi-tasking. There is no reason why this scheduler should be able to schedule the processes on the same processor in an optimal way. The effect is alleviated when the number of processes on each processor is reduced as in the case of two processors per station. Fig.5 shows that if the distribution of processes remains the same but specific priorities are introduced, both the one and two processor stations show significant performance improvement as well as a smoothing of the throughput curves. The receiver process simply gets

lowest priority. In this way the sender is more often able to feed new messages into the communication subsystem, which is therefore optimally used. The receiver is executed only when the rest of the system cannot proceed any further.

One can conclude that prioritising helps the scheduler to optimise the system. Another conclusion is that doubling the number of transputers at each station, from one to two, increases the performance of our system. The speedup is about a factor of two for the non-prioritised system. In the case of the prioritised system most of the performance increase occurs for intermediate message lengths, e.g. for a message length of 32 bytes the speedup is 38%.

Studying the processor load distribution (fig.6) helps to understand the differences in performance for stations with one and two transputers better. The processor load arising from the service user has been plotted relative to the total load of a one-transputer station. Consequently, approximate load balancing between both transputers is achieved for message lengths between 32 and 64 bytes. This compares well with the maximum speedup measured at 32 bytes (see above). The performance is not doubling in this region because there the data link layer spends between 40 and 50 % of the processor time on waiting (see fig.4). For small and large messages the load distribution shows bottlenecks caused by the data link layer or the service user, respectively, which also affects performance.

3.3 Complete system

Here the results for the complete system consisting of data link layer, network layer and service user are described. The corresponding software architecture has been shown in fig.2. Here the number of processors is gradually increased from one to three per station. In the case of the three transputer station each layer has its own transputer. In the case of the two transputer station the network layer and the service user share the same processor. Prioritising has been used throughout this section; the receiver is always assigned the lower priority (The transputer supports only two levels of priority). The checking in the service user has been decreased to avoid a bottleneck in the service user. Only every fourth byte is checked.

The results are presented in fig.7. For the two transputer stations the performance increase is between 40 % for small messages and 30 % for long messages. This moderate performance increase is due to long times spent waiting by the data link layer (see section 3.1). As a consequence, the three transputer station shows no performance increase beyond the two transputer station (not shown in fig.7). Note that the curve for two transputers in fig.7 is almost equal to the curve labelled "real system" in fig.3. This shows that the data link layer is the bottleneck in the complete system.

4 Summary and conclusions

The performance for a multi-layered communication system has been studied for different transputer configurations. With some simplifications (no call connect/disconnect) the message throughput can be analysed stepwise from simple to more complicated configurations:

- The performance of the data link layer has been analysed in detail in terms of the time spent waiting by this layer and also in terms of more primitive processes such as

unidirectional message transfer over "hard" transputer links, the header processing of the data link layer and the communication model of the same system.

- The more complicated system, consisting of a data link layer and a service user augmented by a checking procedure, has been analysed in terms of doubling the number of processors per station as well as using priorities to optimise process scheduling.

- The complete system, consisting of data link layer, network layer and service user, has been analysed in terms of the number of processors used per station.

Prioritising can be useful in an efficient system where many processes share the same processor. Increasing the number of processors gives a performance improvement if the time needed by the processes off-loaded from the processor executing the data link layer is larger than the amount of time the data link layer has to spend waiting. Load charts can be used as a tool for processor load balancing. The best system performance is achieved for long messages. Substantial improvements could be expected from replacing the stop-and-wait protocol used in the data link layer by a sliding window protocol. Avoiding the passing of complete user messages between all layers of the protocol stack, as has been done in this implementation, would probably increase system performance.

References

[At87] Phil Atkin, *"Performance Maximisation"*, INMOS Technical Note 17, Bristol, 1987

[Bb85] Bolt Beranek and Newman (BBN) Laboratories Inc., *"Butterfly Parallel Processor Overview"*, V.1, Cambridge/Mass., June 1985

[Bo88] J.E.Boillat et al., *"Communication Protocols and Concurrency: An Occam Implementation of X.25"*, appeared in: Proceedings, 1988 International Zurich Seminar on Digital Communications, ETH Zurich, p.C2.1 (IEEE Catalog No. 88TH0202-2)

[Bu88] J.W.Burren and D.J.Clarke, *"The Use of Transputers in Digital Communications"*, Proc. 5th Internat. Conf. on Digital Processing of Signals in Communications, IERE, Loughborough, September 1988

[Bu89] J.W.Burren, *"Flexible Aggregation of Channel Bandwidth in Primary Rate ISDN"*, to be published.

[Cl88] D.J.Clarke, *"The Use of Transputers in the Project Unison"*, IEE Colloquium on 'Hardware Support for Communication Networks' (Digest No.83) 7/1-4, Publ.: IEE, London, 1988

[Fr84] Stuart A.Friedberg, *"Architecture and Hardware of the BBN Butterfly"*, University of Rochester, Computer Science Departement, October 1984

[I88a] Inmos Ltd., *"Transputer Reference Manual"*, Prentice Hall, N.Y. 1988

[I88b] Inmos Ltd., *"IMS T425/ IMS T425M Transputer"*, Inmos, Bristol, Sept. 1988

[I88c] Inmos Ltd., *"occam-2 Reference Manual"*, Prentice-Hall, N.Y., 1988

[Is84] International Standards Organization, *"Open Systems Interconnections - Basic Reference Model"*, International Standard, ISO7498, 1984

[Je88] M.N.Jensen, M.Skov, *"Multi-Processor Based High-Speed Communication Systems"*, Proceedings of the Sixth European Fibre Optic Communications & Local Area Networks Exposition, June/July 1988, Amsterdam, 1988

[Po88] Dick Pountain and David May, *"A Tutorial Introduction to occam Programming"*, BSP Professional Books, Oxford, 1988

[Ta81] Andrew S.Tanenbaum, *"Computer Networks"*, Prentice-Hall, Englewood Cliffs, N.J., 1981

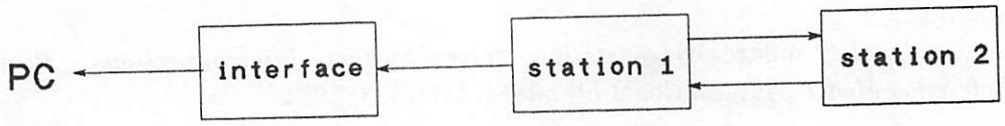


Figure 1: Experimental Architecture

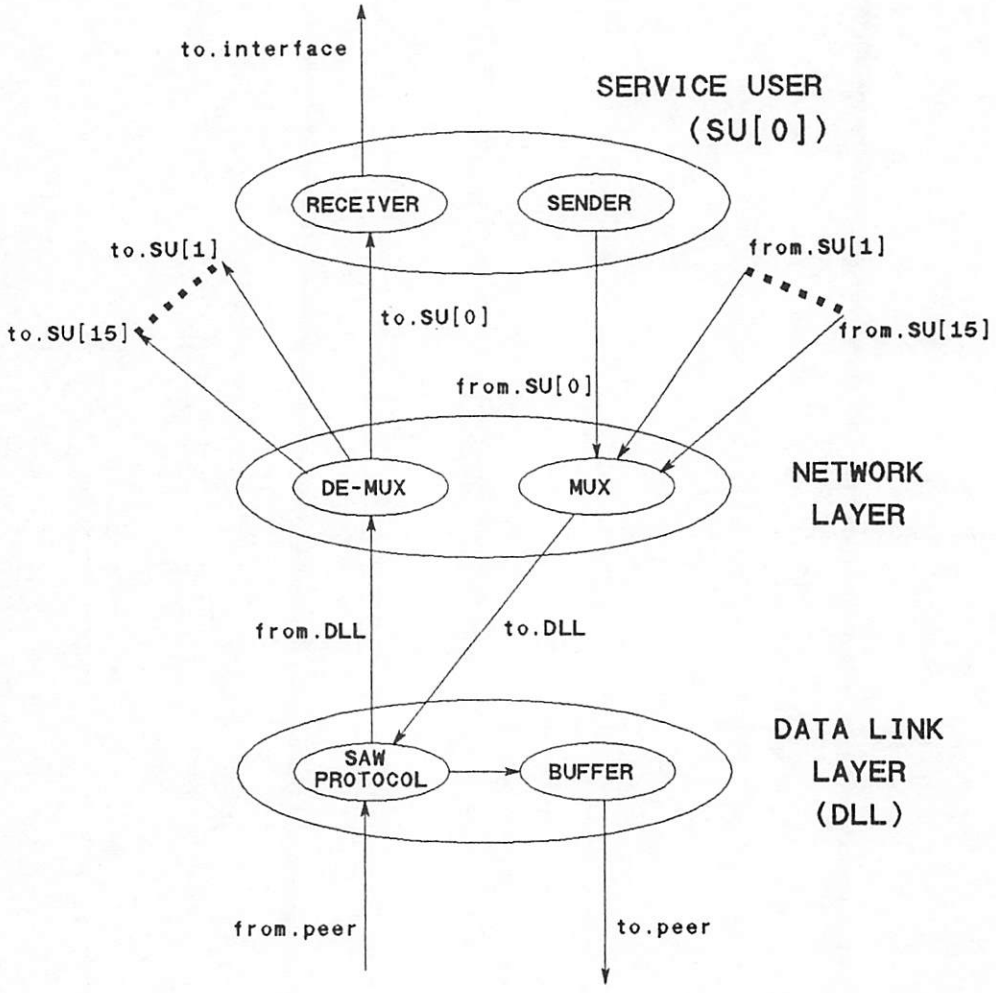
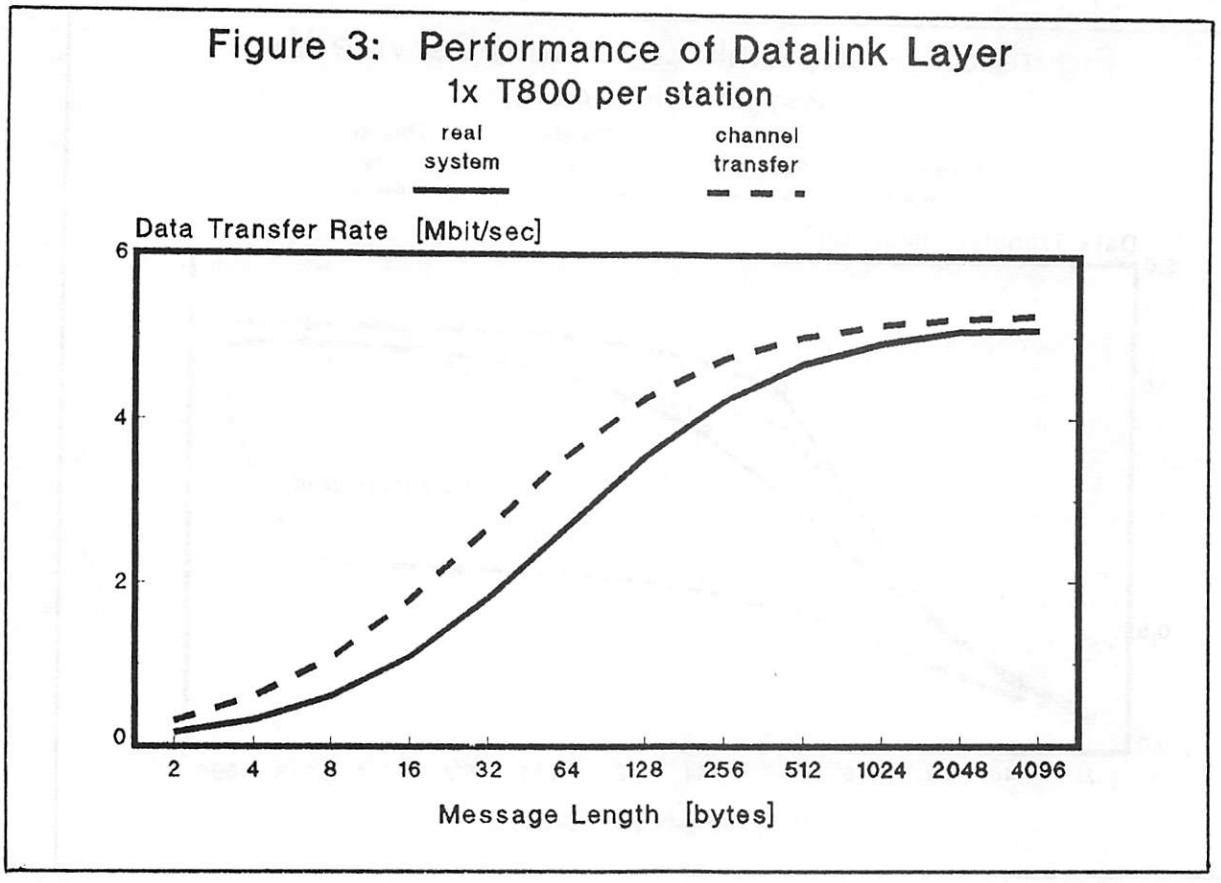


Figure 2: Station Software


Figure 3: Performance of Datalink Layer
1x T800 per station
real system
channel transfer
Data Transfer Rate [Mbit/sec]
6
4
2
0
2
4
8
16
32
64
128
256
512
1024
2048
4096
Message Length [bytes]

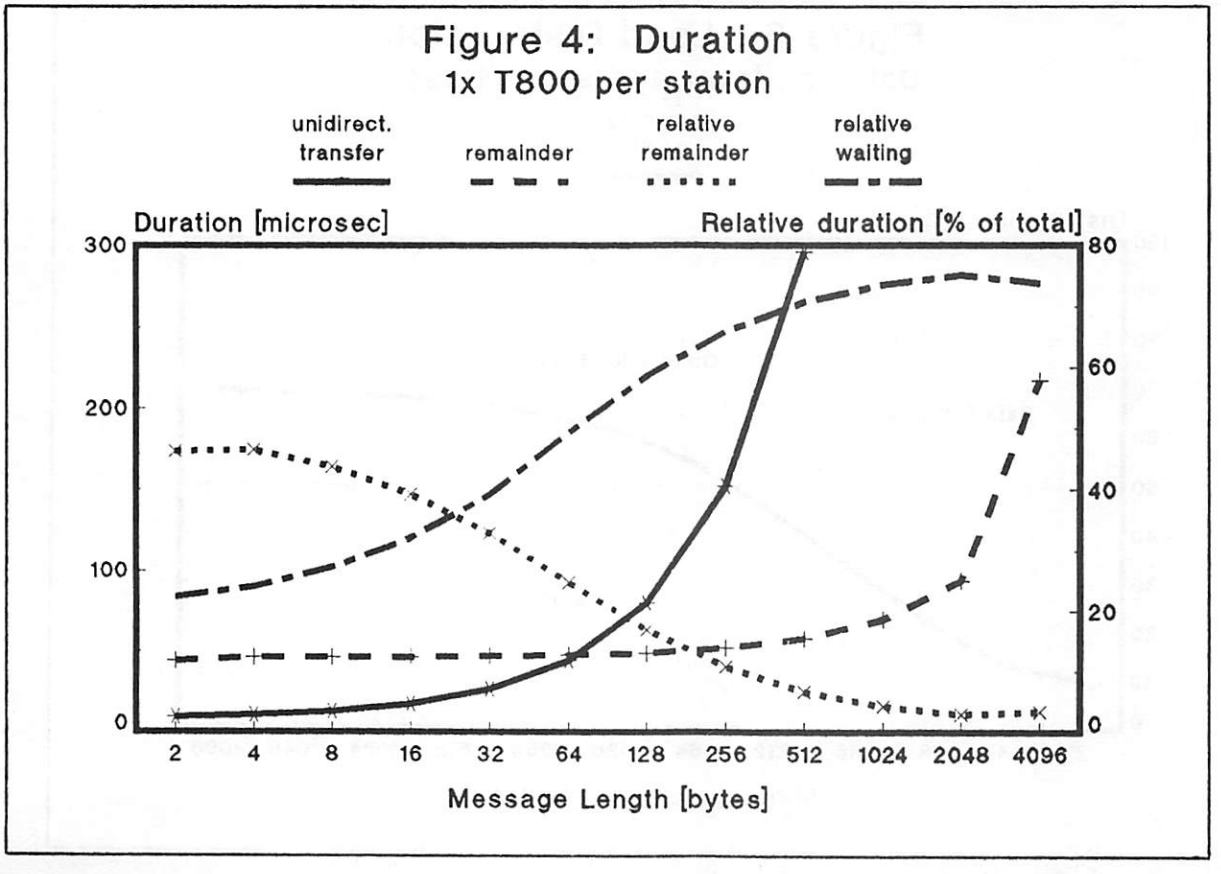

Figure 4: Duration
1x T800 per station
unidirect. transfer
remainder
relative remainder
relative waiting
Duration [microsec]
Relative duration [% of total]
300
200
100
0
80
60
40
20
0
2
4
8
16
32
64
128
256
512
1024
2048
4096
Message Length [bytes]

Figure 5: Data Link Layer and Service User

Asymmetrical case

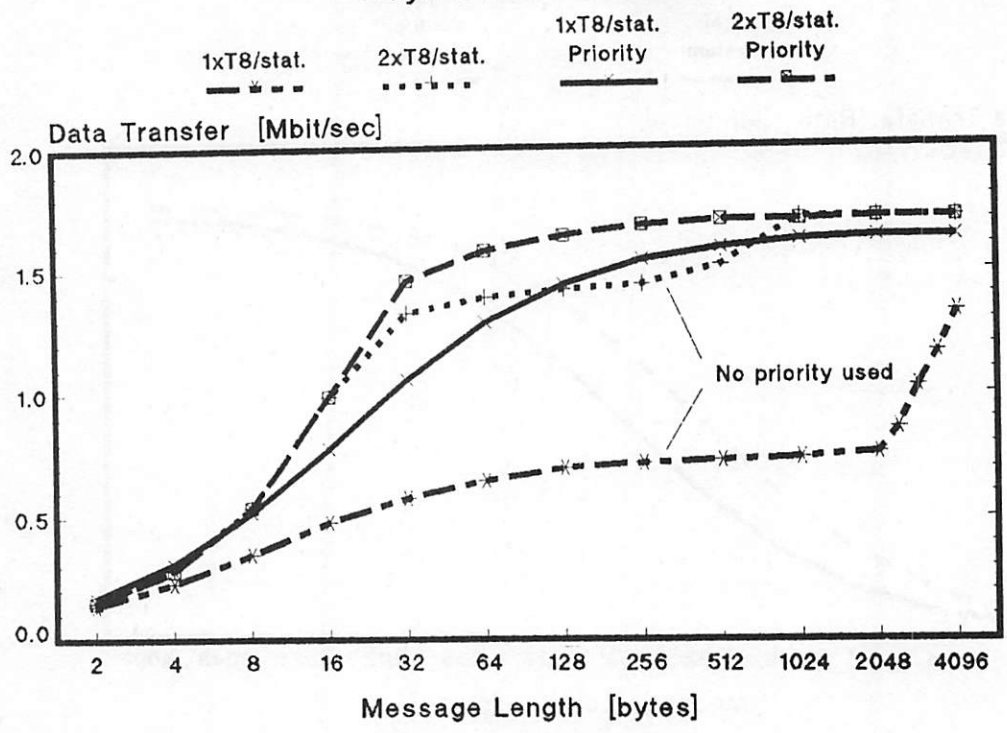


Figure 6: Load Distribution

Data link layer and service user

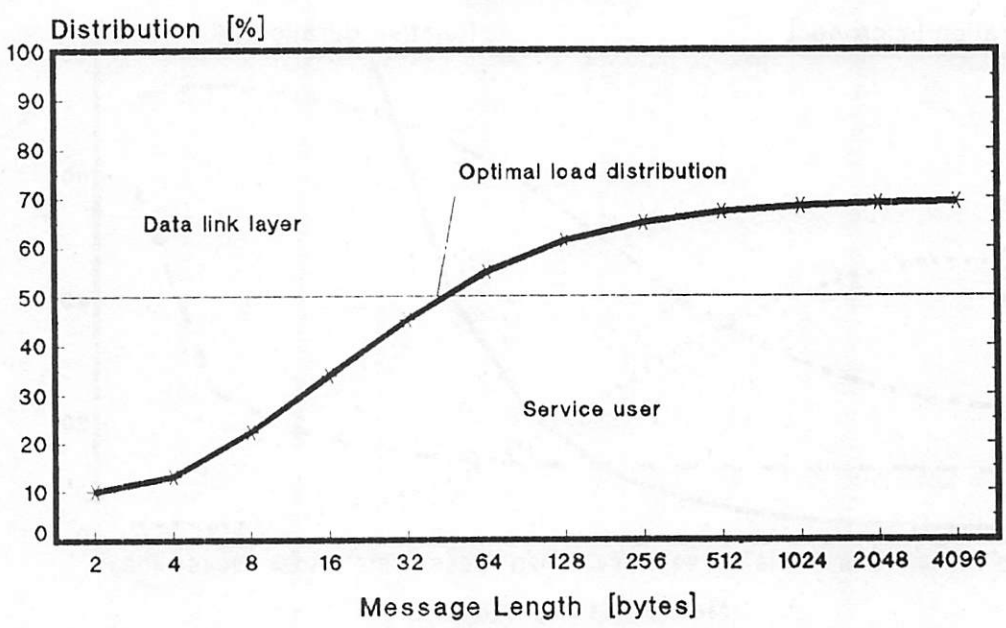
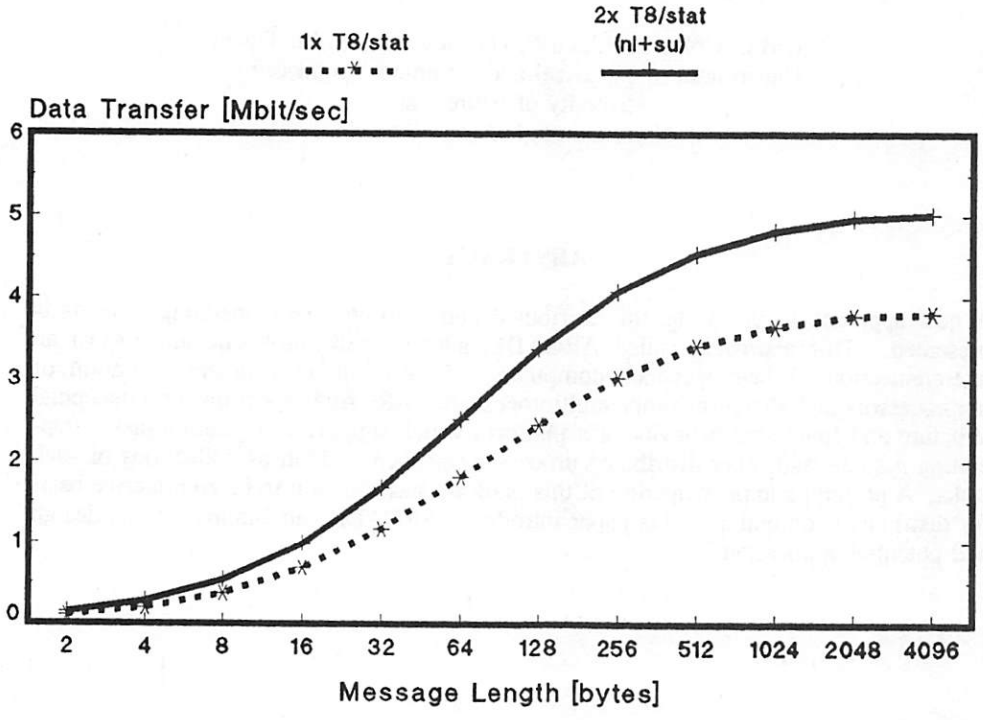


Figure 7: Complete System

Data link layer, network layer and service user



ARCADE:
A Platform for Heterogeneous Distributed Operating Systems

David L. Cohn William P. Delaney Karen M. Tracey
Department of Electrical and Computer Engineering
University of Notre Dame
Notre Dame, Indiana 46556

ABSTRACT

A new approach to the design of distributed and multiprocessor operating systems is presented. This approach, called ARCADE, addresses the problems inherent in an interconnection of heterogeneous computers. It also allows efficient operation of uniprocessors and shared-memory multiprocessors. ARCADE specifies the conceptual structure and functional behavior of a platform which supports cooperating tasks. Operating systems and other distributed processes can then be built as collections of such tasks. A prototype implementation of this platform has shown it to be an effective basis for distributed computing. This paper introduces ARCADE and summarizes its design and potential applications.

1. Goals of the ARCADE Project

The primary goal of the ARCADE project is to provide a software platform on which distributed applications, including operating systems, can be built. An important decision in the design phase of this project was to view applications as collections of independent, cooperating tasks. ARCADE itself is simply the underlying *environment* that supports these tasks and allows them to interact with each other across machine and network boundaries. The most ambitious objective of the project is to provide an environment in which tasks can cooperate in a uniform and effective manner even if they reside on machines with different hardware architectures. This paper describes the experiences of the authors in designing and implementing a system which satisfies these goals.

The ensuing section of this paper summarizes several important projects with goals similar to those of ARCADE. This is followed by an explanation of ARCADE's approach to distributed computing. The next section outlines the major elements of ARCADE and explains why they were developed. The current state of ARCADE, including its implementation, performance characteristics and operating system services, is then discussed. Finally, the future of the project is outlined.

2. Related Work

Much recent research work in the area of distributed computing has focused on object-oriented systems. However, some successful and influential projects have used the more conventional *task-based* model. Since ARCADE uses ideas and concepts that have evolved from both object-oriented and task-based systems, a summary of important projects from both of these areas is presented below. It is followed by

---

This work was supported by a grant from the IBM Corporation under the direction of the Endicott Programming Laboratory.

an overview of the ARCADE distributed environment and the features which distinguish it from related work.

2.1 Clouds

Clouds is an object-oriented distributed operating system developed at the Georgia Institute of Technology [9] [18]. A primary goal of the Clouds project is to provide a highly reliable distributed computing facility for a set of general-purpose computers connected by a local area network. It is based on the object/thread paradigm. In Clouds, objects are the primary abstraction for data storage, while threads provide computational capabilities. Much of the Clouds research is focused on mechanisms for providing an *atomic transaction* facility that guarantees the consistency of all data contained in objects. Clouds has been implemented on VAX processors connected by Ethernet.

2.2 Argus

Argus is an object-oriented programming language and run-time support system developed at the Massachusetts Institute of Technology [16] [17]. Argus is similar to Clouds in that its language component provides the abstractions necessary to shield programmers from the reliability problems of distributed systems. It provides two important program structuring concepts: *guardians* and *actions*. These concepts facilitate the development of applications consisting of distributed components that interact in a serializable and consistency-preserving manner. Argus has been implemented atop the UNIX operating system on a network of MicroVAX machines.

2.3 Emerald

Emerald is a programming language for distributed systems developed at the University of Washington [5] [6]. Like Argus and other similar languages, Emerald is object-oriented. It differs from other such systems, however, in that it presents the programmer with a completely uniform view of objects. That is, an Emerald programmer does not distinguish between small objects (e.g. simple integers) and large ones (e.g. files), nor between local objects and remote ones. In other languages, programmers commonly make such distinctions to achieve acceptable levels of performance and to reduce implementation overhead. In Emerald, however, these distinctions are made by the compiler, which determines an appropriate implementation and invocation technique for each programmer-defined object. A prototype version of this object-oriented system has been implemented on top of the UNIX operating system.

Since all Emerald entities are forced to adhere to a single object model, any object, regardless of its size or complexity, can be migrated from one site to another in a uniform manner. For example, a simple integer may be moved from node to node with the same mechanism used to move a large file object. Emerald's most attractive features are its uniform object model and its support of mobile objects.

2.4 Mach

Mach is an operating system kernel developed at Carnegie-Mellon University [1] [21]. It provides primitive, low-level services for task management and intertask communication. It is highly portable and has been implemented on many different hardware platforms [19].

The Mach kernel supports four fundamental abstractions: tasks, threads, ports and messages. A *task* consists primarily of an address space and access rights to various system resources. It is a passive entity. The active entity defined by Mach is the *thread.* As in most other thread-based systems, multiple threads may execute within the context of a single task. *Ports* are communication channels that provide messaging capabilities for Mach threads. *Messages* are typed data objects that are sent and received via ports.

Mach is, in a sense, a distillation of the important components of the UNIX kernel. It provides a low-level platform on which the bulkier components of a typical UNIX configuration, such as a file system, can be built. It has proven to be an especially effective and elegant UNIX base for tightly-coupled multiprocessor configurations.

2.5 V Kernel

The *V Kernel* is a distributed system base developed at Stanford University [7]. It is similar to Mach in that it provides a set of primitive services with which more sophisticated systems can be built. V supports two primary abstractions: processes and interprocess communication (*IPC*). Software systems built on top of V typically consist of multiple processes which communicate using V's IPC primitives.

Processes are the only active entities in V. The *IPC primitives* provided by V allow processes to interact with each other. These primitives are simple, fast, and effective. A blocking client-server model is used. A sender (i.e. client process) initiates a message transaction and is suspended until a reply is available from the receiver (i.e. server process). Messages are fixed in size (32 bytes), but may contain access rights to variable size segments of the client's address space. A server can use special IPC primitives to copy data to or from these segments.

3. The ARCADE Approach

The approach used in the ARCADE project, as in Mach and V, is to provide a set of primitive services to support tasks and task interaction mechanisms. This approach is used because it seems more fundamental and concrete than the object-oriented approach. For example, the object abstractions of Clouds, Argus and Emerald all require activation and support of task-like entities during the execution of an application program. The run-time support component of each system provides a mapping from the abstract notion of objects to the concrete notion of tasks. By focusing on the concrete, kernel-level aspects of distributed computing, ARCADE facilitates a layered approach to building more sophisticated systems. The valuable abstractions of a typical object-oriented system, for example, can be implemented by a relatively simple run-time support layer that sits on top of the ARCADE platform. Furthermore, the kernel-based approach allows other high-level abstractions to be used concurrently with, and perhaps even in conjunction with, object-oriented systems.

As mentioned above, the conventional kernel-based approach used in ARCADE is similar to both Mach and V. However, ARCADE has some important features that distinguish it from these related systems. First, the services and abstractions that comprise the ARCADE environment layer were designed as a *minimal set*. That is, the environment consists only of those components deemed absolutely essential for effective distributed processing and multiprocessing. During the design process, features were added to the environment layer only if it was impossible or highly impractical to implement them on top of existing features. The result of this minimalist approach was a relatively simple low-level design for which a prototype implementation could be constructed in a straightforward manner. The kernel-level services of Mach were designed using a different philosophy. The Mach kernel provides a much larger set of services that can be used to control low level details of task operation and interaction [3].

Another distinguishing feature of the ARCADE project is its goal of transforming a collection of interconnected (and perhaps heterogeneous) machines into a *seamless* computing facility. In this scenario, tasks interact with each other using uniform semantics, regardless of the configuration of the underlying computer interconnection. For example, two tasks would cooperate using exactly the same semantics if they were located on the same uniprocessor, on different CPUs within a multiprocessor complex, or on different computers connected by a communication network. By providing support for uniform semantics across a variety of computer configurations, the ARCADE approach simplifies the development of applications for distributed and multiprocessor systems. Developers need not worry about low-level details such as communication protocols and translation between incompatible data representations. Rather, these details are handled by the kernel-level software that implements the ARCADE environment.

The abstractions and services defined within the ARCADE environment must be consistent across a variety of processor and network configurations. Consequently, the interface between tasks and the underlying environment could not be specified in an implementation-dependent or machine-specific manner. Rather, ARCADE is an *architectural* definition for a distributed environment. It specifies the conceptual structure and functional behavior of the environment as seen by a task. The details of the architecture are described in the following section.

4. Description of the Architecture

This section describes the conceptual structure and functional behavior of the ARCADE distributed environment. The architectural specification defines two principal abstractions: tasks and data units. The structure of tasks is presented first. This is followed by a discussion of the task synchronization and control mechanisms provided by the environment. The characteristics of ARCADE data units are then described, along with the architecturally-defined mechanisms for transferring and sharing data units between tasks. This section concludes with a summary of the ARCADE services for task creation, exception handling, and error recovery.

4.1 Task Structure

An element of execution must be defined in any low-level operating system specification. In ARCADE, the basic element of execution is called a *task*. The architectural specification for the ARCADE environment identifies several structural components of a task. These components are illustrated in Figure 1 and described below.

ARCADE's notion of an *address space* is similar to that used in conventional systems. A task's address space consists of the entire range of memory addresses that can be specified by the task. However, in the ARCADE environment, it is more appropriate to view an address space as a collection of memory segments into which data units can be mapped. As illustrated in the figure, the ARCADE environment manages a pool of data units which are not considered part of the task structure. Data units will be fully described later.

Various mechanisms are provided by ARCADE to allow a task to request that particular data units be mapped into its address space. One of these mechanisms involves the task *input queue*. An input queue is similar to a Mach port [3], but is not defined as a separate abstraction as in Mach. Rather, one input queue is associated with each ARCADE task. The input queue is a first-in first-out collection of *notification packet data units*, or *NPDUs*. When one task transfers a data unit to another task, the environment creates an NPDU and places it at the end of the destination task's input queue. The destination task can then use the NPDU to gain access to the transferred data unit.

In order for tasks to cooperate, they must be able to identify each other. Therefore, each ARCADE task has a unique character string called a *logical name*. Tasks use logical names to identify each other when they interact. These names are constructed in a hierarchical fashion. Furthermore, they are completely location-independent. That is, a task's name does not include any indication of the machine on which the task is running. This feature allows a transparent task migration facility to be defined as part of the architecture.

Several other task components are illustrated in the figure. The environment maintains a *child list* for each task so that its children can be located. The *unique identifier*, or *UID*, is used as a shorthand task identification tool. The *security level* and *privilege vector* combine to provide a foundation for a trusted operating system. They make it possible to guarantee that all potential security violations are detected, prevented, and reported. The *task state* includes transient information, such as register contents, which must be maintained in a multitasking environment. The remaining components of a task are associated with ARCADE's control and synchronization facilities; these are discussed next.

4.2 Control and Synchronization

Task interaction mechanisms are critical components of distributed and parallel systems. Some systems use messaging for all interaction. Frequently, this leads to purely synchronous cooperation. Standard remote procedure calls, where the caller blocks until the procedure completes, are typical of message-based interaction. In order to provide a basis for various types of distributed systems and applications, ARCADE needed a more flexible control and synchronization scheme. Support for both synchronous and asynchronous cooperation was required. Therefore, a new approach to control and synchronization, which combines the notions of interaction, suspension and termination, was devised.

The ARCADE mechanism for control and synchronization is more uniform and comprehensive than those used in other distributed systems. In Mach, for example, a task may be suspended or terminated in a

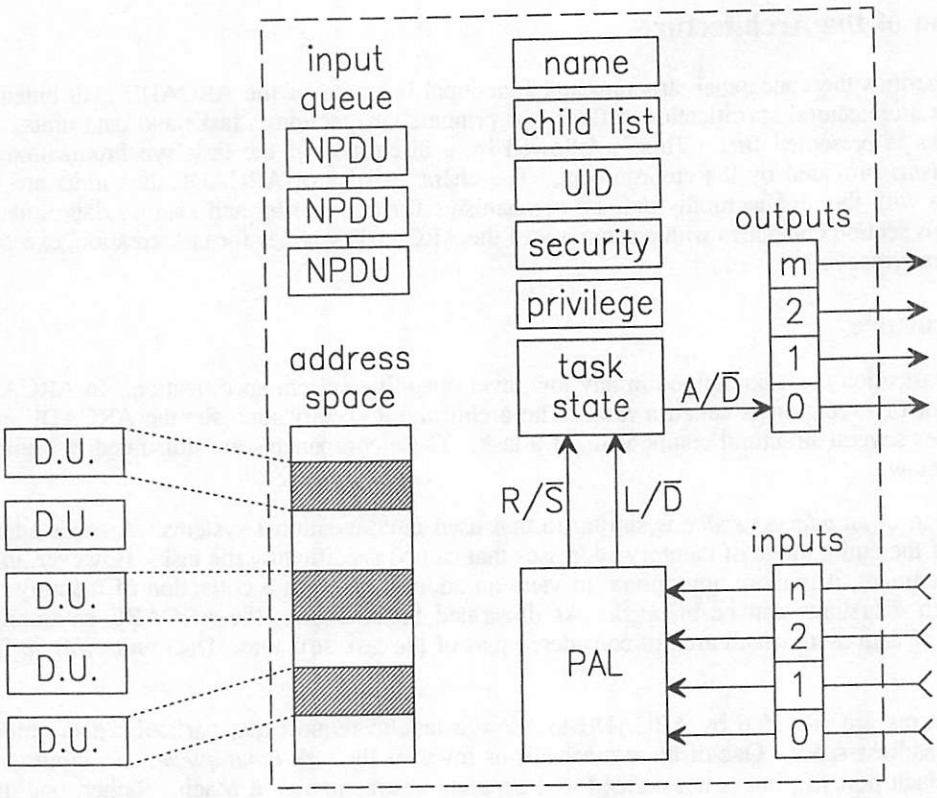


Figure 1. ARCADE Task Structure

variety of ways using various kernel-level services [3]. In ARCADE, however, the suspension and termination mechanisms are more streamlined. ARCADE's model is based on the simple notion of binary-valued inputs and outputs, coupled with a programmable array logic (*PAL*) component implemented in kernel-level software. The relationship between these three components and a task's state is illustrated in Figure 1.

Every ARCADE task has a set of *outputs* that may be used to influence other tasks. A task can set the value of individual outputs to either ON or OFF. One output, however, is controlled by the environment on behalf of the task. This is called the *ALIVE/DEAD* output. It remains ON only while the task is alive.

By manipulating its PAL and inputs, a task can allow itself to be influenced by the outputs of other tasks. A task may request that outputs of another task be attached to its inputs. After attaching such outputs to its inputs, the task specifies how its inputs are to affect its operation.

A task uses its PAL to specify exactly how attached outputs are to influence it. Each input is fed into the PAL. The PAL, in turn, generates two independent *control values* using Boolean functions of the inputs. The *RUN/SLEEP* control value indicates the task's desire to run. When it is ON, the task is dispatched in the normal fashion. Conversely, when it is OFF, the task is suspended. The *LIVE/DIE* control value indicates the task's desire to continue to exist. If the LIVE/DIE value ever switches to OFF, the task is terminated by the environment.

The Boolean functions used to generate the control values are specified, or programmed, by the task using an environmental service. Thus, by programming the PAL and attaching output signals to the appropriate inputs, a task can specify the exact manner in which external events are to influence its operation. Furthermore, the PAL allows tasks to respond to composite events. For example, a task might attach several outputs to its inputs. Its PAL can then be programmed so that the task sleeps until some arbitrary combination of values is present at the inputs.

The control and synchronization facility can also be used to service interrupts. ARCADE defines a set of *pseudo-outputs* associated with processor interrupts. When the ARCADE kernel determines that an

interrupt is being presented to the processor, it pulses the pseudo-output associated with the interrupt. An interrupt handler task can attach such a pseudo-output to one of its inputs. It can then program its PAL so that it sleeps until an interrupt occurs. Upon waking, it processes the interrupt and then goes back to sleep.

4.3 Data Units

As noted previously, the address space of ARCADE tasks contains mappings to *data units*. Most activity in a typical ARCADE system involves the manipulation of data units by groups of cooperating tasks. Since all machine boundaries are transparent, special provisions must be made to ensure that data can be transferred properly between tasks residing on heterogeneous machines. For this reason, the ARCADE environment treats data units as structured carriers of data. That is, the type of data within each data unit is known by the environment.

This approach is a direct consequence of the desire for transparent machine boundaries between heterogeneous computers. If data is to be transparently transferred, the environment must be responsible for any necessary translations. For example, two machines might use different representation conventions for floating point numbers. The environment can only perform the necessary translation between representations if it knows the type of data that is to be transferred.

A data unit comes into existence when a task explicitly requests its creation, or allocation. When requesting allocation of a new data unit, a task informs the environment of the data unit's structure via a type specification. As in high-level languages such as Pascal and C, an ARCADE type specification is a structured combination of simple types. The set of simple types supported by the ARCADE environment includes normal data types and two new pointer-like types. These simple types may be combined into arrays and records, just as in a language with user-defined types.

Pointers present special difficulties when data is to be transferred between tasks on different machines. Normally, a pointer's value is a machine address. This makes little sense, however, on the destination machine. Thus, ARCADE supports a special pointer-like type called an *offset*. Offsets allow dynamic data structures to be constructed within a single data unit. Unlike a conventional pointer, an offset's value is not the machine address of its target. Rather, it is the *displacement* of the target, in bytes, from the base of the data unit. As stated above, the ARCADE kernel is responsible for constructing a replica of a data unit on one machine when it is needed by a task that resides there. The values of any offsets within such a data unit can then be adjusted to account for memory alignment constraints on the entities within the data unit.

Since an offset is defined to be a displacement from the base of a data unit, the target of an offset must reside in the same data unit as the offset itself. A second special pointer-like data type is defined in ARCADE to circumvent this restriction. It is described below.

4.4 Data Unit Links

The ARCADE environment supports dynamic structures consisting of multiple data units by providing a special pointer-like type called a *data unit link*. Like other simple types, data unit links can be specified as components of a data unit. However, unlike all other simple data types, data unit links cannot be directly manipulated by tasks. Rather, tasks must use environmental services to manipulate them.

The *SETLINK* service allows a task to assign a value to a data unit link. An example of this service appears in Figure 2. Before requesting the service, the task has access to two data units, labeled A and B. It then requests that the data unit link inside A (represented by a solid circle) be assigned a target of data unit B. As shown in the figure, the environment simply establishes a record of the fact that a target has been assigned to the link.

A task uses the *ACCESS* service to request that the target of a data unit link be mapped into its address space. An example is presented in Figure 3. In the figure, the task has access to data unit A, which contains a link to data unit B. After completion of the ACCESS operation, data unit B has been mapped into the task's address space by the environment.

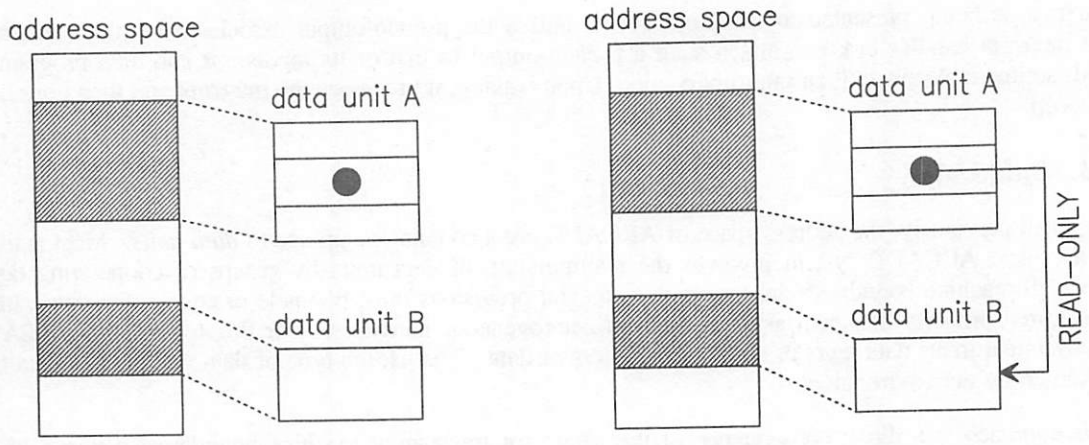


Figure 2. Before (L) and After (R) SETLINK

ARCADE's data unit links possess some novel features which make them attractive in a distributed environment. Most importantly, data unit links can transparently span machine boundaries. For example, before the task of Figure 3 accesses data unit B, the machine on which the task is running might not even possess a copy of B. When the ACCESS service is requested, the task's local ARCADE kernel can transparently obtain a replica of the target data unit. The requesting task cannot distinguish between operations that require remote communication and those that are handled locally.

Second, the data unit concept allows construction of arbitrarily complex, multi-level data structures. The concept is general enough to support *chains* of data units connected by links. Few existing distributed systems offer such flexibility. Mach, for example, supports *typed messages* in which only a single level of pointers can be defined [3].

Since all manipulation of data unit links is performed by the ARCADE environment, the environment can maintain its own record of each data unit link's value. The actual contents of data unit link fields can be ignored. This prevents malicious or runaway tasks from corrupting the data unit management subsystem.

The practice of specifying and retaining data type information at run time is not unique to ARCADE. The *Agora* programming system, for example, uses a similar approach [4]. Agora, like ARCADE, provides transparent translation of structured data during interactions between heterogeneous machines. Agora also supports complex, hierarchical data structures. The unique aspect of the ARCADE approach, however, is its ability to support rigidly structured memory using only *conventional* address space management practices. Agora, on the other hand, provides special abstract primitives for establishing the address mappings needed to access shared data structures [4]. Because of its highly conventional nature, ARCADE could be used as a foundation for the more abstract mechanisms used in the Agora system.

4.5 Data Unit Transfer and Sharing

An important benefit of ARCADE's data unit structuring capabilities is the ease with which both simple and complex data structures can be transferred between tasks. The same mechanisms are used regardless of whether the tasks share a uniprocessor, use shared-memory multiprocessors, or run in a distributed system. In each case, the kernel can provide efficient, transparent data management.

ARCADE provides three data transfer services: *MOVE*, *COPY* and *SHARE*. The MOVE service allows a data unit to be removed from the address space of the source task and added to the address space of the destination task. COPY is used to create a new data unit with the same structure and content as the source data unit. The duplicate is then added to the address space of the destination task. SHARE makes possible the illusion of shared memory between tasks even when no physical shared memory exists.

The ARCADE data transfer services make extensive use of the data unit link abstraction for both basic transfer operations and transfers involving complex, multi-level data structures. Implementation of each service is based on the following four-step approach:

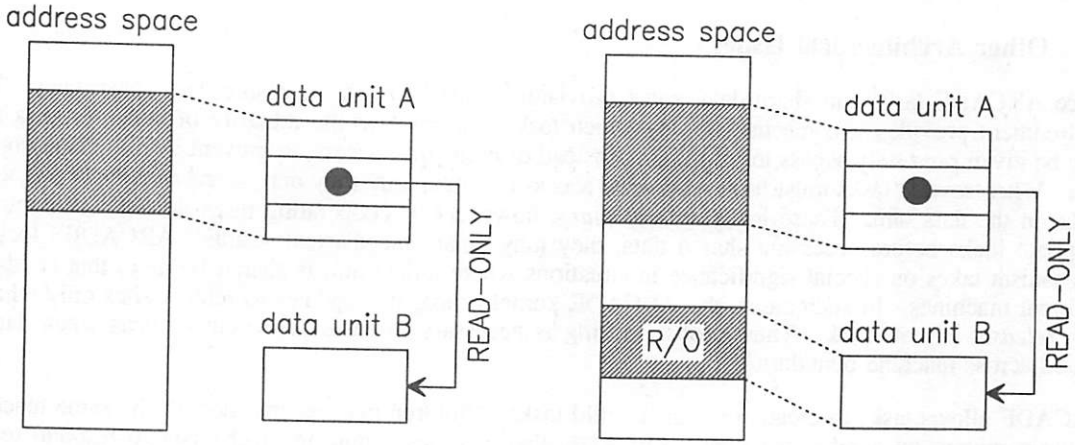


Figure 3. Before (L) and After (R) ACCESS

1. When a task requests one of the data unit transfer services, the environment builds a *notification packet data unit*, or *NPDU*. The NPDU contains information about the transfer, such as a time stamp, the identity of the source task, and the type of service used in the transfer. In addition, one of the fields in an NPDU is a data unit link. The environment sets the NPDU's link to point at the data unit being transferred.

2. The NPDU is sent to the destination machine (if it is remote) and added to the input queue of the destination task.

3. The destination task uses the *RECEIVE* service to gain access to the NPDU. The environment moves the NPDU from the task's input queue into its address space. The NPDU is then directly addressable by the task.

4. Finally, using the data unit link inside the NPDU, the destination task can access the data unit that was transferred by the source task. The standard ACCESS service is used in this final step.

By using the same basic mechanism for each service, ARCADE is able to present a uniform and coherent set of data transfer options for cooperating tasks.

The MOVE service is essentially a messaging service. However, the use of data units allows efficient messaging in shared memory systems and preserves normal messaging between distributed computers. COPY was added to ARCADE to minimize overhead when a replica of the data was needed.

The ability to share data units between tasks, even those that reside on different machines, is a unique feature of ARCADE. Some researchers question the effectiveness of this abstraction [21]. Of course, the ability to share memory among tasks within a single machine or in a multiprocessor with physical shared memory is clearly desirable. Therefore, if ARCADE is to be effective in such situations, it must provide data unit sharing services. However, given the design objective of supporting transparent machine boundaries, all ARCADE services must support uniform semantics for both local and remote operations. Consequently, ARCADE allows data units to be shared between any tasks, even if they reside on different machines.

The implementation of an inter-machine shared data facility is clearly a difficult chore. Yet, this abstraction has several benefits. Most importantly, it can simplify the development of distributed application programs. The developers of the Agora programming system discovered that the shared memory paradigm often provides a natural and elegant model on which to base distributed applications [4]. A related research effort involved the analysis of a *problem-oriented shared memory* paradigm [8]. Again, the conclusion was that shared memory, even across machine boundaries, provides an appropriate conceptual base upon which to build distributed applications. Thus, despite the costs associated with implementing this abstraction, shared memory can provide significant benefits by simplifying the development of multi-machine application programs.

4.6 Other Architectural Issues

Since ARCADE tasks can share data, some provisions must be made to ensure data consistency. The environment provides two mechanisms by which tasks can preserve the integrity of data. First, a task may be given *read-only* access to a data unit, instead of *read/write* access, to prevent it from changing the data. When several tasks must have read/write access to a data unit, they may assert *read locks* and *write locks* on the data unit. These locks are *voluntary*, however. If cooperating tasks do not assert the appropriate locks before accessing shared data, they may obtain inconsistent results. ARCADE's locking mechanism takes on special significance in situations where a data unit is shared by tasks that reside on different machines. In such cases, the ARCADE kernel propagates updates to remote sites only when a task *releases* a write lock. Thus, proper locking is necessary to ensure coherent replicas when data is shared across machine boundaries.

ARCADE allows tasks to create, or *spawn*, child tasks. Children may be spawned on the same machine as the parent or on another machine. ARCADE also provides a built-in mechanism to respond to exceptions and failures.

4.7 Prototype Implementation and Performance Evaluation

A prototype of the ARCADE architecture has been implemented and is undergoing extensive test and evaluation. The primary objectives of the prototype implementation are:

- Clarify the architecture
- Verify its feasibility and practicality
- Ensure that it does not impose undue performance constraints

The prototype environment is not yet a complete implementation of the ARCADE architecture. However, it addresses all of the goals listed above. The major components of the architecture, including the basic task model, synchronization mechanisms and data unit management facilities, are fully operational. Implementation issues associated with the remaining components, such as lock management, have been carefully studied and evaluated. This section summarizes the design and implementation techniques used for the major components of the architecture. It concludes with some preliminary performance figures for the prototype.

The current prototype implementation of the ARCADE environment is called ARCADE/x86. It is based on Intel 80286 [14] and 80386 [15] microprocessors (*80x86*) and IBM Personal System/2 computers [11]. Although the prototype system currently supports only a single type of machine, it includes all of the provisions necessary for operation in a heterogeneous environment. As noted in Section 5, the implementation of a truly heterogeneous system is currently in progress.

The 80x86 has several features which make it an attractive platform for ARCADE. First, it provides built-in task management facilities, including a multi-level task privilege hierarchy and a fast, hardware-based task dispatching mechanism. It also supports segmented address spaces. Each task's address space can contain up to 16,384 variable-size segments. These characteristics mesh well with ARCADE's basic task and data unit concepts. ARCADE tasks are implemented using the 80x86 task support facilities and ARCADE data units use the 80x86 memory segmentation scheme.

The prototype implementation supports up to fifteen individual nodes running ARCADE/x86. The nodes are connected by an IBM Token Ring Network [12] [13]. The Token Ring's 4 Mbps data rate provides excellent communication performance. Also, the adapter card's firmware includes physical layer, data link layer, and network layer protocols. Currently, all network communication in the prototype environment is handled by this subsystem.

Code for the prototype ARCADE kernel was written almost entirely in C. Since standard C compilers exist for most, if not all, conventional processors, the prototype kernel is essentially portable. However, the current implementation makes heavy use of the segmented address space capabilities of the 80x86 processor. Porting to another segmented architecture, such as IBM's System/370, should be straightforward. Porting to a machine with a flat address space, on the other hand, may require significantly more work. Further details of the prototype implementation are available in [10].

As noted above, the objectives of the prototype implementation were three-fold. The first was to resolve any ambiguities in the original design of the architecture. Although some minor modifications were made in the original specification, it essentially survived the early test intact.

The feasibility study and the performance analysis objectives are closely related. For example, the implementation of a particular component of the architecture is feasible only if the performance of the resulting code is acceptable. Thus, these two objectives are analyzed together in the following paragraphs.

The preliminary performance analysis focuses on three distinct components of ARCADE: the control and synchronization mechanism, the data unit management subsystem, and the data unit transfer facilities. All of the services provided by these components exhibit uniform semantics for both local and remote operations. Thus, a reasonable approach to performance analysis is to determine and compare the time required for entirely local and entirely remote operations.

All of the performance figures cited below were gathered using IBM Personal System/2 Model 70-A21 computers operating at 25 MHz. For operations requiring network transactions, the prototype kernel's communication subsystem was used to drive the IBM Token Ring Network Adapter/A at 4 Mbps. All tests were run under minimal loading conditions.

The performance test for the control and synchronization subsystem was carried out by first attaching the output signal of one task to the input signal of a second task. The second task's PAL was then programmed such that the task was forced into the SLEEPING state until the attached signal was switched ON. The test results indicate the delay between the time the output signal's value was switched ON and the time the sleeping task started to run again.

As shown in Figure 4, a wake-up operation involving two local tasks requires only 600 microseconds in the prototype environment. However, when the two tasks are located on different machines, the delay increases to 19 milliseconds. The obvious reason for the increase is the required communication between the two tasks' kernels when the output signal's value changes. Despite the dramatic difference between the local and remote cases, a 19 millisecond delay is reasonable for tasks that are located on different machines.

The ARCADE ALLOCATE service is used to generate new data units, while the SETLINK service allows existing data units to be linked together in a dynamic fashion. Performance characteristics for both of these service routines are shown in Figure 4.

Allocating a new data unit requires approximately 950 microseconds in the prototype environment. Setting a data unit link requires 250 microseconds. These figures are much larger than the figures for equivalent operations, namely buffer allocation and pointer assignment, in a conventional system. However, this is to be expected; the figures are quite reasonable considering the advantages of ARCADE's data management approach.

A task uses the ACCESS service to request that a data unit (which is the target of a data unit link) be mapped into its address space. The task specifies a data unit link and the environment finds the target data unit and performs the mapping. Uniform semantics are used regardless of whether the data is local or remote.

Type of Operation	Local	Remote
Task Wake-up	600 microseconds	19 milliseconds
Data Unit ALLOCATE	950 microseconds	Not Applicable
SETLINK Operation	280 microseconds	Not Applicable
Data Unit ACCESS	830 microseconds	See Figure 5
Data Unit Transfer	1.7 milliseconds	18 milliseconds

Figure 4. Performance Figures for the Prototype Environment

The performance test for data unit ACCESS operations measured the time required to process an ACCESS service request. As shown in Figure 4, accessing a data unit that already resides on the local machine is a reasonably fast operation, requiring only 830 microseconds. Furthermore, since local ACCESS operations require no data movement, the performance figure is independent of the size of the data unit being accessed.

Accessing a remote data unit is much more expensive. Inter-machine data transfer operations are necessary because the requesting task's kernel must obtain a local replica of the data unit. Thus, the service time depends on the size of the accessed data unit. The dependence is illustrated in Figure 5. For small data units, the processing time is well below 100 milliseconds. The slight nonlinearity in the dependence on data unit size is caused by inefficiency in the communication subsystem, which currently divides a single large transfer into multiple 8 kilobyte transfers.

The data unit transfer subsystem makes heavy use of the input queue abstraction. For each data transfer operation, ARCADE generates an NPDU and adds it to the input queue of the destination task. The destination task must then receive the NPDU from its input queue into its address space. Finally, the task accesses the actual data unit through the data unit link in the NPDU.

The performance analysis for this subsystem provides a measure of the time that elapses between the initiation of a data transfer operation (e.g. MOVE or SHARE) and the final receipt of the NPDU into the destination task's address space. The task used to carry out this analysis was configured to SLEEP until a data unit was available on its queue. After waking up, the task issued a RECEIVE request. The results shown in Figure 4 indicate that, for an entirely local transfer operation, the steps listed above require a total of 1.7 milliseconds. However, when the source task transfers a data unit to a remote task, the steps require 18 milliseconds. Although these figures are reasonable, efforts are currently underway to improve the performance of data unit transfers in the prototype environment.

Close comparison of the ARCADE performance figures with those of other systems is impractical for several reasons. First, each of the related distributed systems discussed here supports its own unique set of services and abstractions. Furthermore, the hardware platforms and communication facilities vary widely from one system to another. A very rough comparison, however, shows that ARCADE's performance figures are in the same approximate range as those for the Argus and Clouds distributed systems. For those systems, typical remote operations require on the order of 40 to 300 milliseconds [9] [16].

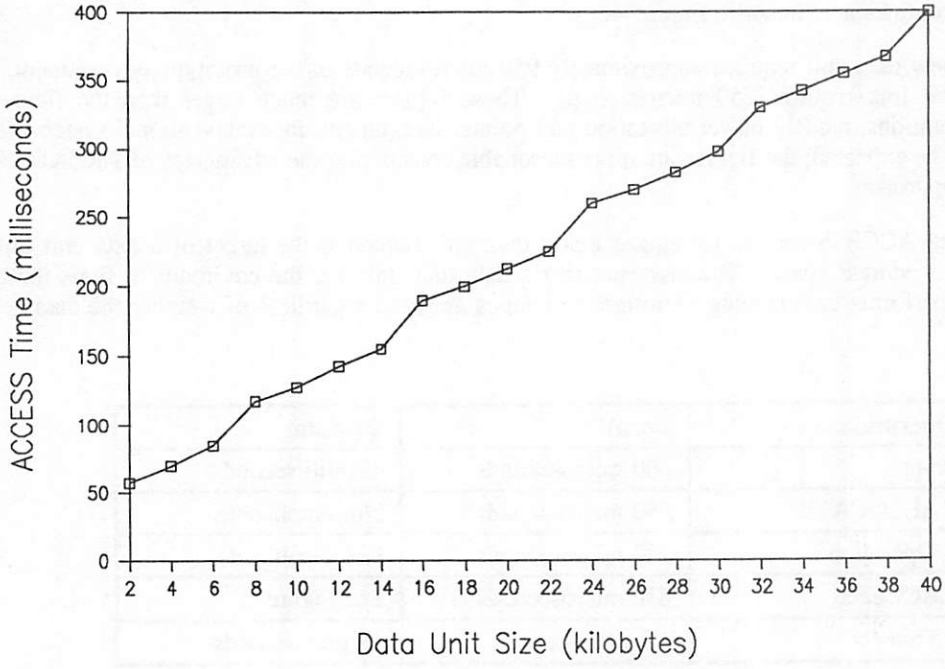


Figure 5. Performance of Remote Transfer Operations

4.8 Operating System Services

The ARCADE architecture is only a *platform* for an operating system. It does not include normal operating system services, such as a user interface or a file system. In order to evaluate ARCADE as a platform, operating system services had to be added to the ARCADE implementation. Since ARCADE/x86 runs directly on the hardware, it was not practical to adapt an existing operating system to the ARCADE environment. Therefore, a custom operating system called *KOS* was developed concurrently with ARCADE/x86 [22]. This section outlines KOS and presents some performance information.

KOS provides many of the high level operating system components that are not part of the ARCADE environment. KOS includes a simple user interface, a file system, and program loading facilities. These operating system components are implemented as a set of six ARCADE tasks:

- TIMER - timed services provider
- DISKTASK - disk device driver
- CONMAN - console manager
- FILESYS - file system
- COMMAND - command interpreter
- LOADER - program loader

The first three tasks manage physical devices: the timer, disk, and console (screen and keyboard). Other tasks which wish to access one of these devices must do so by interacting with the manager of the device. Each manager makes use of the ARCADE pseudo-output associated with the interrupt line for its device. CONMAN updates the screen by writing to video memory. Mapping of video memory into CONMAN's address space is accomplished by an implementation-specific I/O service routine provided by ARCADE/x86.

Many of the tasks (FILESYS, CONMAN, LOADER, TIMER) are server tasks. In general, servers interact with clients through data unit exchanges. ARCADE's control and synchronization facilities are utilized as well. C library routines are provided to make these client/server interactions transparent to an application programmer. Thus, for example, a programmer need not know the details of the interaction with FILESYS. Standard high level file services such as *open* and *close* may be called instead.

The implementation of operating system services in KOS is rather unconventional. However, in many cases, the functions provided by KOS are similar to those found in conventional operating systems such as DOS or OS/2. For example, the services provided by FILESYS mimic those found in the OS/2 file system. Also, COMMAND supports many DOS commands. The similarities between the KOS environment and standard operating systems has allowed users to quickly adapt to the prototype environment.

However, there are significant differences between KOS and the DOS and OS/2 operating systems. Most notably, the KOS services are generally available on a network-wide basis. Because KOS services are provided by ARCADE tasks, a client task may address a KOS server residing anywhere in the ARCADE interconnection. This allows:

- Location dependent remote file access. The KOS file naming convention includes an optional machine name prefix; this allows a user to identify the machine where a file is stored.
- Remote program execution. When running programs, a user can specify that the new task be spawned on a remote machine.
- Remote console access. A program task running under KOS can address any console server in the ARCADE interconnection. Thus, a task executing on one machine can display output on the screen of another machine; it can also receive input from the keyboard of the other machine.

Although KOS is a fairly simple operating system, its implementation illustrates the suitability of the ARCADE architecture as a platform for operating system development. Furthermore, the concurrent development of KOS and ARCADE/x86 demonstrated the usefulness of ARCADE's uniform service interface for both local and remote task interactions.

During early development of ARCADE/x86, the kernel did not support network communication. Therefore, most of the initial KOS components were built and tested within a single machine. When commu-

nication support became available, the KOS servers were able to interact with remote clients in the same manner as with local clients. No changes in the interaction method were required; the only alterations to KOS involved:

1. Appropriate server identification by the client, since many servers were suddenly accessible. For example, to allow remote file access, the file naming convention was extended to include an optional machine name prefix.

2. Regulation by servers of a remote task's ability to access local data or services. For example, the file system task was extended to allow the local user of a machine to restrict the rights of remote tasks to access or modify local data.

Thus, when communication support became available in ARCADE/x86, KOS was quickly transformed from a single machine operating system to one which provided distributed services across a network. Ease of distribution is one of the major benefits of using ARCADE as a platform for the development of distributed applications. In the case of KOS, the "application" is actually the high level components of an operating system.

There are, however, drawbacks to the ARCADE platform approach. The strict layering between the ARCADE/x86 kernel and the KOS operating system components introduces some performance penalties that would not be present in an integrated operating system. For example, KOS takes as much as two to four times longer to copy files than DOS does. Several factors, including the layered structure, are probably responsible for this result. For example,

- Neither ARCADE/x86 nor KOS have been optimized. Instead, functionality has been given the highest priority.

- In general, there is higher overhead involved in the multi-tasking protected mode of the ARCADE/x86 environment as compared to the single-threaded real mode of DOS.

Sometimes it is possible to use ARCADE's flexibility to overcome its poorer performance. Consider a user at machine A that wishes to copy a large file from one directory on machine B to another directory on machine B. There are two ways of accomplishing this:

1. The user can submit a COPY request to COMMAND on machine A. COMMAND in turn will ask FILESYS on machine B to open, read, write, and close the appropriate files. In this case, the file data makes a round trip across the network from FILESYS on machine B to COMMAND on machine A and back to FILESYS on machine B.

2. Here, the data round trip is avoided. The user submits a COPY request to COMMAND on machine *B*. Now, FILESYS and COMMAND are on the same machine. The data is not sent over the network, and, as shown in Figure 6, the operation may be completed in less time than it would take using DOS and a LAN program.

This example is just one illustration of how the advantages of the ARCADE approach can overcome the drawbacks. In the case of high-level operating system components, it appears that the performance penalties of the ARCADE approach are generally outweighed by ARCADE's benefits.

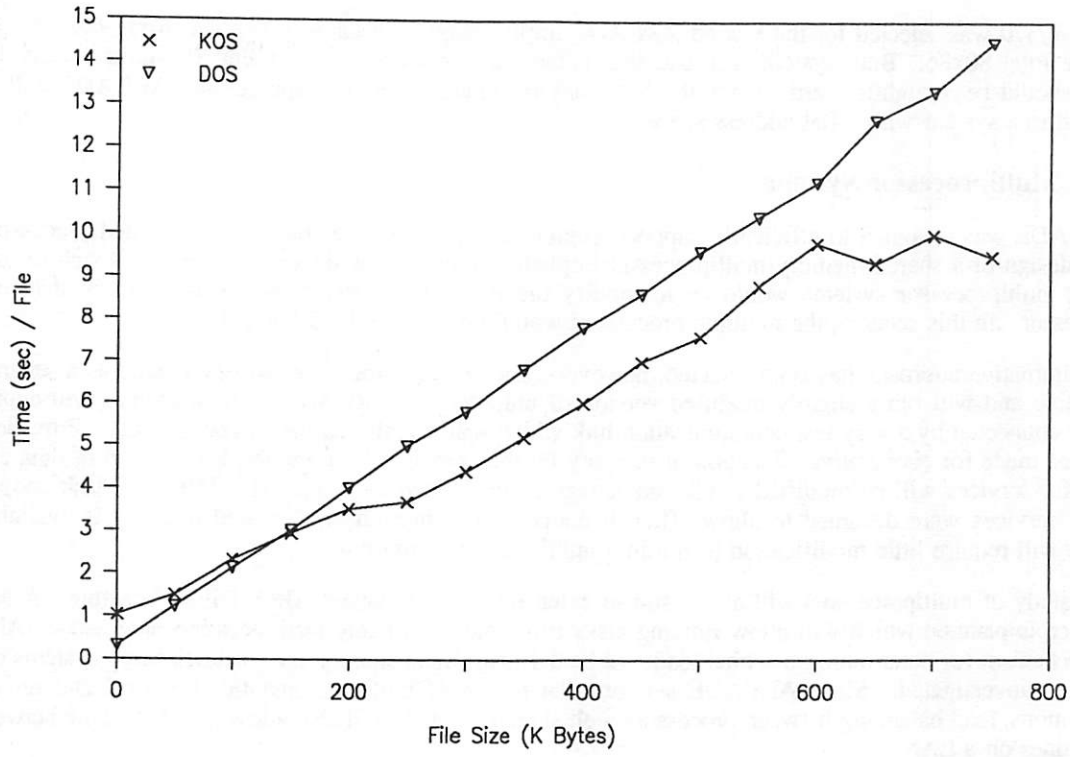


Figure 6. KOS and DOS Times for Remote to Remote Copy

5. Future Work and Expected Benefits

Considerable future work is planned for the ARCADE project. This section describes work currently underway to port ARCADE to additional machines, to handle multiprocessor systems, to provide additional local operating system contexts, to support object-oriented language environments, and to build useful distributed applications.

5.1 Heterogeneous Machine Architectures

Currently, ARCADE runs only on the Intel 80x86 family of processors. However, ARCADE was designed specifically for networks of *heterogeneous* machines. To demonstrate the full benefits of ARCADE's design, the kernel must be ported to other architectures. A port to the IBM System/370 architecture is now in process.

This second ARCADE implementation will highlight the usefulness of the data unit concept. For example, Intel machines use ASCII for character representation, while System/370 uses EBCDIC; Intel and System/370 machines store integers in the opposite byte order; the two architectures also use different floating point formats. However, the data unit concept will allow a task running on an Intel machine to transfer or share data with a task on a System/370 machine as easily as with a task on another Intel machine. The ARCADE kernel will transparently handle all necessary data translation.

The initial translation scheme for this heterogeneous environment will be susceptible to some information loss. For example, translation between the different floating point formats will surely result in lost precision. Furthermore, the Intel floating point format, which conforms to the IEEE 754 Standard, supports the notions of *NaN*s and positive and negative infinity. System/370, on the other hand, provides no such support. Consequently, an application program which performs floating point operations in such an environment may yield different results depending on the particular machines which operated on the data. The various benefits of heterogeneous interoperation, however, are expected to outweigh such anomalies.

System/370 was selected for the second ARCADE implementation because it bears a strong resemblance to the Intel 80x86. Both systems can use segmented address spaces to represent data units. Thus, the port should be straightforward. Once the System/370 implementation is operational, ARCADE will be ported to a system with a flat address space.

5.2 Multiprocessor Systems

ARCADE was designed to efficiently support uniprocessors, multiprocessors, and distributed processors. The design of a shared-memory multiprocessor implementation is now underway. One approach to handling multiprocessor systems would be to modify the ARCADE dispatcher to support more than one processor. In this scheme, the multiple processors would not be visible to the tasks.

An alternative approach has been selected, however. Each of the processors will be treated as a separate machine and will run a slightly modified version of uniprocessor ARCADE. The machines will appear to be connected by a very fast communication link which will actually be the shared memory. Provisions will be made for cooperative allocation of memory by the kernels. Also, the implementation of data unit transfer services will be modified to take advantage of the shared memory. The ARCADE task cooperation services were designed to allow efficient cooperation when physical shared memory is available. They will require little modification to run in a multiprocessor environment.

The study of multiprocessors will also result in extensions to the current ARCADE architecture. A new service is planned which will allow running tasks to be moved, or migrated, between processors. Also, a mechanism for determining machine load will be defined. Load sharing and load balancing systems can then be investigated. Since ARCADE supports the notion of uniform semantics for local and remote operations, load balancing between processors with shared memory will also allow load balancing between machines on a LAN.

5.3 Heterogeneous Local Operating Systems

In addition to supporting heterogeneous machine architectures, ARCADE was designed to allow different local operating system contexts within an interconnection of computers. The local operating system for a particular machine can thus be tailored to that machine's architecture and use. A mainframe, for example, can run a large multi-user operating system while a personal workstation can run a simpler single-user operating system with a sophisticated graphical user interface. If the systems are connected and both are running ARCADE, components of a distributed application could be executing on the different machines. The various components could effectively cooperate using ARCADE services, despite the different local operating system contexts.

Currently, ARCADE only supports the KOS local operating system. KOS was developed in conjunction with ARCADE and provides a limited set of important services. The porting of ARCADE to System/370 will require some modifications to KOS; KOS will have to be altered to support the new hardware. Some of its tasks may need only slight alteration, but others will probably require major modifications.

Work is currently underway to develop a POSIX (IEEE Standard 1003.1) compatible operating system for ARCADE/x86. Like KOS, it will be implemented as a set of cooperating tasks. Early work on POSIX has indicated that an important addition must be made to the ARCADE architecture. To support POSIX signals, external events must be able to cause interrupts within a task. Therefore, the next version of ARCADE will include an asynchronous input queue mechanism as well as the synchronous one discussed earlier.

Once POSIX is operational, it will be possible to evaluate ARCADE's compatibility with full UNIX implementations. This is only one of a set of compatibility studies that are anticipated. Others include OS/2 and VM/CMS.

5.4 Programming Language Environments

The ARCADE environment was designed to facilitate distributed programming. Architectural features such as a consistent task model, implicit communication services and data translation solve many of the problems of constructing distributed applications. However, distributed programming in ARCADE is still a complex affair. The process can be simplified by adding programming language support for distribution.

Other distributed systems, including Argus, Cronus [20], Eden [2], Mach, and Emerald, have shown the value of this approach.

In keeping with current trends, an object-oriented programming language for ARCADE is being developed. Since C is currently supported for use in ARCADE, the language is based on extensions to C. It is being implemented as a preprocessor, much like the C++ programming system. However, the language extensions being developed for ARCADE use a coarse-grained object model in which objects are implemented as tasks. This approach differs from the fine-grained model used in the C++ environment.

The encapsulation and messaging of object-oriented programming are very similar to the ARCADE task model and data transfer services. Tasks will be used to implement objects, while method invocation will be implemented with data unit transfers. The preprocessor will shield programmers from the details of ARCADE, providing instead familiar object-oriented primitives. Recent developments in concurrent object-oriented programming will be used to take advantage of the concurrency inherent in ARCADE.

Although many of the object-oriented paradigms are available in the existing ARCADE environment, an important one is missing. No provisions have been made for *inheritance*. However, an extension of the ARCADE architecture to allow linking of "code units" would permit inheritance.

This work will simplify the process of writing distributed applications. Also, it is the first step in designing a complete programming environment for ARCADE. A custom C compiler and an appropriate linker are currently being built. These will provide linguistic support for exception handling, load balancing, and process migration.

5.5 Distributed Applications

ARCADE was designed to be a platform for distributed applications. One major distributed application currently under construction is a distributed file system. This system will consist of a set of cooperating tasks on top of ARCADE. It will act as a bridge between the heterogeneous local file systems.

While the KOS file system includes some distributed characteristics, such as the ability to access remote files, it is *not* a full-fledged distributed file system. Features such as file replication and location transparency are not supported. Upon completion of the System/370 port and the POSIX implementation, two more local file systems will be supported in the ARCADE environment. Initially, these systems will not provide distributed services. Each will be designed to manage only local files.

A true distributed file system will allow any task in an ARCADE interconnection to use any file in the interconnection as if it were stored in the local file system format. On way to build such a system would be to scrap the existing local file systems and replace them with a new system. However, an attractive alternative is possible in the ARCADE environment.

The distributed file system will be built as a distributed application. The cooperating tasks which comprise the file system will act as clients of the existing local file systems. Thus, the local file systems will remain intact. The tasks which make up the distributed file system will keep track of replicated files and file locations. If a task requests access to a replicated file, the distributed file system tasks will determine which copy of the file to use and provide access through the appropriate local file system.

6. Acknowledgements

The authors would like to thank Timothy M. Bryden, Arindam Banerji and Edward R. Smith for their contributions to this paper. Special thanks also go to the members of the research staff in the Personal Computer Laboratory at the University of Notre Dame for their many contributions to the ARCADE project.

7. List of References

[1] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of Summer Usenix*, July, 1986.

[2] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D. "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January, 1985, pp. 43-59.

[3] Baron, R., Black, D., Bolosky, W., Chew, J., Golub, D., Rashid, R., Tevanian, A. and Young, M. "MACH Kernel Interface Manual", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December, 1987.

[4] Bisiani, R. and Forin, A. "Multilanguage Parallel Programming of Heterogeneous Machines", *IEEE Transactions on Computers*, Vol. 37, No. 8, August, 1988, pp. 930-945.

[5] Black, A., Hutchinson, N., Jul, E. and Levy, H. "Object Structure in the Emerald System", in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, Portland, OR, October, 1986, pp. 78-86.

[6] Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January, 1987, pp. 65-76.

[7] Cheriton, D. R. "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, April, 1984, pp. 19-42.

[8] Cheriton, D. R. "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design", in *Proceedings of the 6th International Conference on Distributed Computing Systems*, IEEE, Cambridge, MA, June, 1986, pp. 190-197.

[9] Dasgupta, P., Leblanc, R. and Appelbe, W. "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", in *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE, San Jose, CA, June, 1988, pp. 2-9.

[10] Delaney, W. P. "The ARCADE Distributed Environment: Design, Implementation and Analysis", Ph.D. Dissertation, University of Notre Dame, Notre Dame, IN, April, 1989.

[11] *IBM Personal System/2 Hardware Interface Technical Reference*, IBM Corporation, Part No. 68X2330, 1988.

[12] *IBM Token Ring Network Adapter Technical Reference*, IBM Corporation, Part No. 69X7713, 1986.

[13] *IBM Token Ring Network Architecture Reference*, IBM Corporation, Part No. SC30-3374-01, 1987.

[14] *Intel iAPX 286 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, Order No. 210498-002, 1984.

[15] *Intel 80386 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, Order No. 230985-001, 1987.

[16] Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. "Implementation of Argus", in *Proceedings of the 11th Symposium on Operating System Principles*, ACM, Austin, TX, November, 1987.

[17] Liskov, B. "Distributed Programming in Argus", *Communications of the ACM*, Vol. 31, No. 3, March, 1988, pp. 300-312.

[18] Pitts, D. and Dasgupta, P. "Object Memory and Storage Management in the Clouds Kernel", in *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE, San Jose, CA, June, 1988, pp. 10-17.

[19] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Transactions on Computers*, Vol. 37, No. 8, August, 1988, pp. 896-908.

[20] Schantz, R. E., Thomas, R. H. and Bono, G. "The Architecture of the Cronus Distributed Operating System", in *Proceedings of the 6th International Conference on Distributed Computing Systems*, IEEE, Cambridge, MA, May, 1986, pp. 250-259.

[21] Tevanian, A. and Rashid, R. F. "MACH: A Basis for Future UNIX Development", Report CMU-CS-87-139, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June, 1987.

[22] Tracey, K. M. "The Design and Implementation of an ARCADE-based Operating System", Master's Thesis, University of Notre Dame, Notre Dame, IN, April, 1989.

A Decentralized Embedded Operating System Supporting Distributed Execution of Ada Tasks

Roger K. Shultz
Commercial and Government Avionics Division
Rockwell International
Cedar Rapids, Iowa

ABSTRACT

The problems, challenges and results of implementing a decentralized embedded operating system are presented, with discussion of design methods and the use of Ada tasking for operating system implementation.

A Prototype, Reconfigurable, Integrated System for Multiprocessing (PRISM) has been developed for future fully integrated avionics systems. The PRISM provides a fault tolerant and damage tolerant network of processor, memory, and I/O resources. PRISM is the platform for demonstration of decentralized embedded operating system concepts including support for distributed Ada tasking as an application programming interface.

The PRISM inter-processor communications and control structures are designed specifically to support the execution and communication needs of concurrent Ada tasks. The PRISM Network Operating System serves as a decentralized network manager, allowing distributed application tasks to communicate without prior knowledge of the system configuration. The Network Operating System is itself composed of concurrently executing Ada tasks. Both the Network Operating System and the application programs utilize the Ada tasking model for scheduling, synchronization, and inter-task communication.

An overview of the multiprocessing system architecture is presented with a discussion of the requirements and support for the distributed Ada programming system. The capabilities of the network and local operating systems are given, in addition to trade-offs in PRISM operating system design and implementation. Distributed rendezvous timings from experiments with multiple PRISM clusters show the effect of alternative operating system implementations that both restructure the system tasks and re-distribute system functions.

1.0 INTRODUCTION

A distributed multiprocessor computer architecture created for fully integrated avionics systems is serving as a decentralized embedded systems research platform for the Processor Technology Department of Rockwell International Corporation. The Rockwell prototype, reconfigurable, integrated system for multiprocessing (PRISM) architecture is designed to provide a fault tolerant and damage tolerant network of processor, memory, and I/O modules. The system is composed of hardware building blocks configurable into a wide variety of system architectures.

Each PRISM cluster is a processing site for the decentralized *network operating system*. Each processing element in a cluster executes a copy of the *local operating system*. Together, replicated network and local operating systems provide fault tolerant, distributed embedded system support.

An underlying datagram mail system stores and forwards packets throughout PRISM. In this manner, multiple message channels are supported between subsystems of the network operating system and local operating system. Built into this communication system is voting based detection and management of faults. Supported by this communication system, application tasks perform fault tolerant distributed Ada rendezvous.

The PRISM architecture is designed to be implementation technology independent. That is, processing elements, memory elements, and communication channels may be implemented with any technology meeting system requirements. The architecture is designed for maximum flexibility, allowing it to realize a wide range of system throughput, reliability, size, weight, and power requirements.

PRISM is managed by a decentralized system controller allocating functions to the distributed processing sites, performing system level redundancy management, reconfiguration, and collecting system state knowledge. Task scheduling and other local processing site services are handled by the local operating system. The decentralized network operating system links the Local Operating Systems together for inter-task communications, global data accesses, and global I/O.

High level language programming is of utmost importance in applying a complex system implementation to embedded avionics applications. The system services of PRISM are tailored to support requirements of Ada, and to facilitate the efficient execution of Ada tasking operations. This approach allows a distributed PRISM computer system to be entirely programmed using standard Ada language constructs.

Section 2.0 ARCHITECTURAL DESCRIPTION summarizes the PRISM system hardware components. More detailed PRISM architecture descriptions are found in other publications [Best 88, McGahee 88]. Section 3.0 PRISM OPERATING SYSTEM describes its capabilities, including distributed Ada requirements and restrictions. Design trade-offs between two operating system implementations and their affect on throughput are discussed in Section 4.0 SYSTEM IMPLEMENTATION TRADE-OFFS. Section 5.0 CONCLUSIONS contains a summary of the important points covered in this paper. All referenced publications are listed in Section 6.0.

2.0 ARCHITECTURAL DESCRIPTION

The general organization of a generic PRISM multiprocessing system is to group individual processing units, memory elements, and I/O units into a processing cluster (also referred to as a *node*), allowing the processing units to share common resources and providing a level of localized control. These multiprocessor clusters are then networked over parallel or serial buses to form a distributed embedded system.

The PRISM architecture defines a set of hardware building blocks: processors, memories, interfaces, and controllers. A Network Operating System, coupled with the system communications channels, combine these building blocks into a coherent fault tolerant distributed processing system.

The architecture supports redundancy of both system hardware and software. Redundant elements are loosely synchronized, permitting physical separation and dissimilar redundancy, including the application of *N-version programming techniques* [Avizienis 85]. Furthermore, the system architecture does not impose re-

quirements on the individual processing element architecture, allowing systems to be implemented using dissimilar processor redundancy to compensate for implicit hardware and software faults.

The PRISM approach for implementation of Distributed Ada tasking parallels the *virtual node* approach in DIADEM [Atkinson 88]. Distributed tasking entry calls are implemented through a message passing protocol between nodes. A communication network supports inter-node communication. DIADEM defines virtual nodes, but in PRISM these are nodes specifically designed into the hardware architecture. As in DIADEM, PRISM nodes execute application code which does not share global data with other nodes. Also, inter-node communication is restricted to entry calls.

Designing nodes into the hardware architecture allows fault management transparent to distributed applications. Although it can be argued that the actions taken upon processor failure should be part of the application [Knight 87], hardware support of fault management is most efficient. Communication redundancy and message comparison voting are part of the PRISM hardware design. The operating system shelters an application from faults by greatly reducing the probability that all copies of an application task fail.

Processing clusters are physically separated to provide damage tolerance. Application software is executed redundantly where several clusters perform the same operations and compare results to detect faults. To provide fault recovery and graceful degradation, software tasks are migrated between processing elements.

Global data objects are stored in global memory elements that are distributed among the processing clusters. Redundant global memory elements are physically separated for improved reliability and damage tolerance. Global data objects are migrated between global memory elements to provide fault recovery.

The PRISM architecture was developed to support system programming with Ada. As such, PRISM system communication resources are organized to support the Ada language for implementing multitasking application software. Application tasks are distributed among the processing clusters, with all of the communications and decentralized control functions handled by the network operating system.

2.1 Processing cluster architecture

In order to minimize the amount of hardware required to implement a multiprocessing system, processors share common resources. To provide fault containment, localized areas of configuration control act as *fire walls* against fault propagation. The PRISM architecture provides fire walls by combining processors into clusters. In general, a processing cluster contains a number of computation, memory, and I/O elements.

A cluster block diagram is shown in Figure 1. A PRISM cluster contains seven modules: the Processor Memory Element (referred to as a *processor*), Cluster Controller Module (referred to as a *cluster controller*), Regional Memory Element (referred to as *regional memory*), Global Memory Element, Global Bus Interface Unit (referred to as the *bus interface*), Redundancy Management Unit (referred to as the *redundancy unit*), and I/O Interface Units. Figure 1 illustrates simplex, dual-re-

dundant and quad-redundant communication channels. The operating system provides redundancy scaling for messages routed between channels.

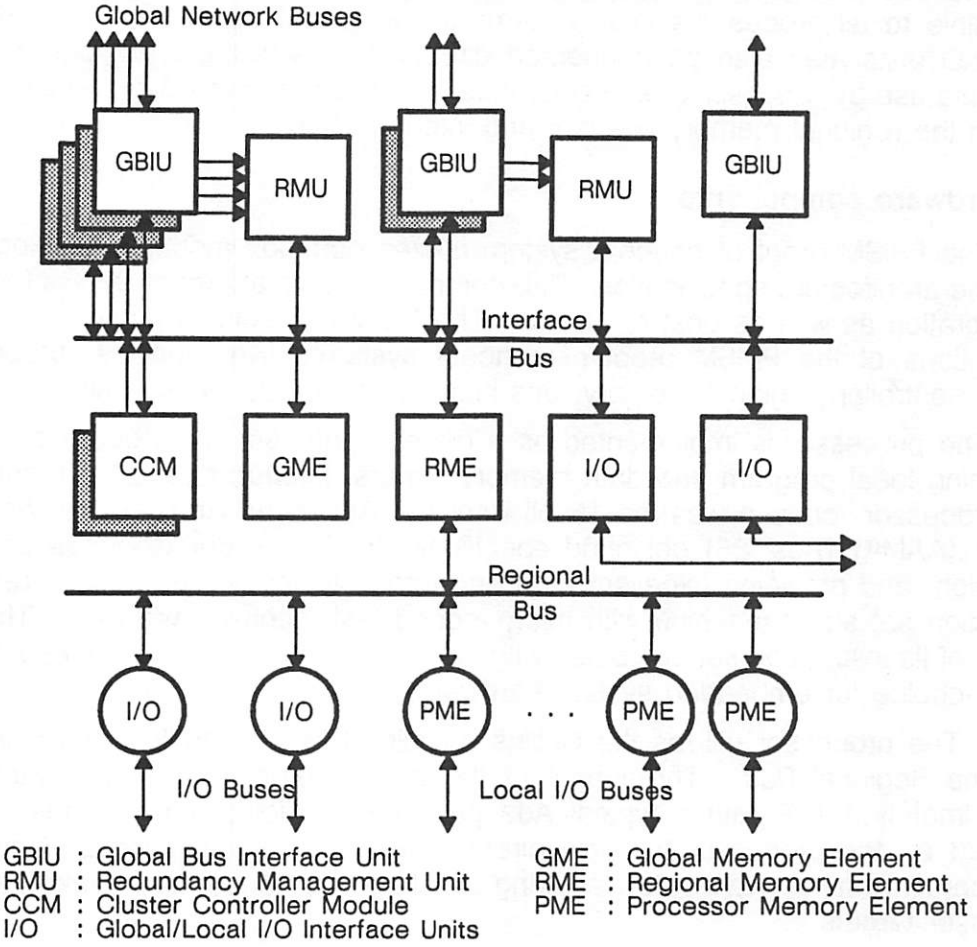


Figure 1. PRISM cluster composition

A cluster controller is responsible for interfacing the processing elements within the cluster to the global communication network linking together various branches of the architecture. All communication between processing elements is handled by the cluster controllers. In the event of faults, a cluster controller isolates failed modules by disabling individual processors.

Global Memory Elements are incorporated to provide distributed global data storage. Global Memory Elements are accessed over the Global Communication Network, providing common storage shared by processing clusters.

Within a processing cluster, common regional memory and I/O resources are accessed through a shared regional bus. Generally the Regional Bus will be implemented as a parallel bus to maximize bandwidth to shared resources. Sharing of the regional bus presents a point of contention between the processors within a cluster.

Processors within a cluster have access to the global communication network through one or more bus interfaces. This network provides the means for system control and configuration, task communication and synchronization, and global data object access.

External data may be accessed globally or within one node. Global I/O buses are accessed through the use of global I/O interface units. Global I/O buses are accessible to all processors in a system via the global communication network. Local I/O units may also be connected directly to the cluster's regional bus for dedicated use by processing elements, thus reducing communication requirements through the regional memory element and interface bus.

2.2 Hardware components

The PRISM proof of concept system utilizes common modules developed to meet the architecture specification. The common module approach allows for ease of integration as well as cost reduction. The following sections provide functional descriptions of the PRISM proof-of-concept system's five modules: processor, cluster controller, regional memory, bus interface and redundancy unit.

The processor is implemented as a general purpose single board computer containing local program and data memory, timers, interrupts, and I/O resources. The processor chosen was the 16-bit Rockwell Advanced Architecture Microprocessor (AAMP) [Best 85] designed specifically for high order language program execution, and providing excellent compiled code efficiency. AAMP is a reduced instruction set stack machine with micro-coded task context switching. The efficiency of its instruction set combined with a simplified bus interface makes AAMP a logical choice for embedded system hardware.

The processor utilizes the Collins Avionics Bus for parallel communication over the Regional Bus. Throughput of the current processor implementation is approximately 1 MIP with a typical Ada program instruction mix. A test port is included on the processor to allow direct connection of the Rockwell Computer Development Station [Lyttle 85], allowing simultaneous access to as many as four processor targets.

The hardware for the cluster controller is identical to that of the processor. Initial consideration was given to program and data memory capacity as well as interrupt capabilities so that a common module could serve each function.

The regional memory is a 256K (16-bit) word random access memory with an 8K byte memory map. An Interface Bus port is provided to the cluster controller for random access of the full 256K word block, as well as to the memory map for configuration of the Regional Bus port. Regional Bus accesses are translated based on unique processor identifier keys, and the state of the memory map. This mapping technique provides the cluster controller with the capability to assign protected mailbox areas to each processor as well as common memory areas which are accessible by multiple processors.

The bus interface is implemented as a 16-bit message bus which utilizes self-timed arbitration and data transfer techniques. The bus interface provides separate master and slave interfaces required for redundant operation where the transmitting node is one of *n* redundant nodes. The bus interface is capable of autonomous transmission of a block of data to the selected destination receiver(s). Receivers respond to four message destination types: hardwired address, programmable (redundant) address, broadcasts local to a particular global bus, and system wide broadcasts.

The redundancy unit provides autonomous comparison and voting of message data on either a single word basis or through a direct memory access (DMA) block transfer mechanism. A comparison is initiated by the cluster controller supplying the redundancy unit with bus interface port addresses and corresponding data page addresses of messages to be compared.

2.3 Global communication

Processing clusters are combined together to form a network of processor, memory, and I/O resources. Processing clusters interface to the Global Communication Network through common I/O resources shared by all processors within a cluster. The optimal configuration of network communication paths is dependent on the system application.

A cluster tree network, shown in Figure 2, is often considered a useful configuration. Tree architectures directly support hierarchical control where each subtree is monitored and controlled by its root node. In many applications, a tree architecture is generally good for minimizing both the communication hardware and transport delay of messages between nodes. A network architecture of this type is desirable in that time-multiplexed bus technology can be applied (such as MIL-STD-1553). It is also beneficial in that root nodes may contain resources that are shared by the leaf nodes, permitting a further reduction in hardware.

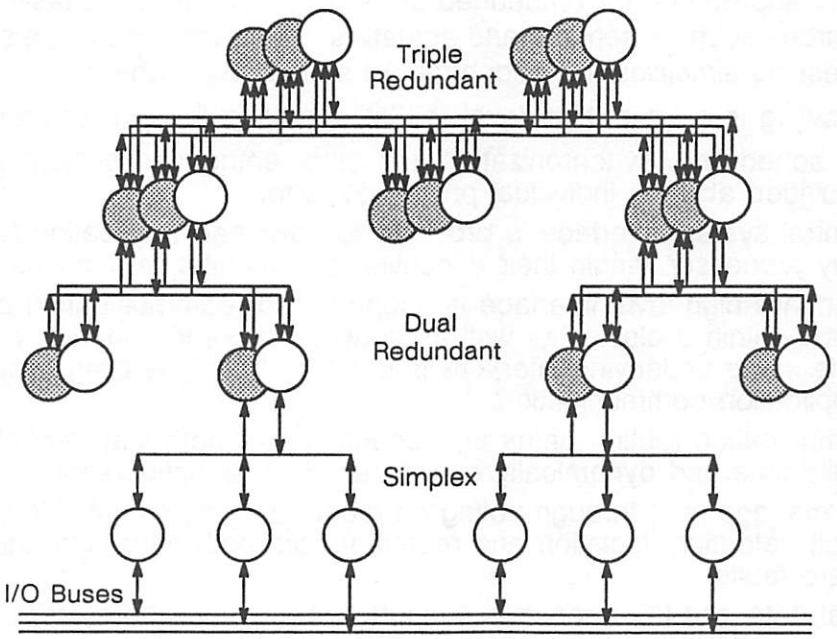


Figure 2. A tree structured PRISM network

As illustrated in Figure 2, redundancy scaling techniques may be applied to satisfy fault tolerance, damage tolerance, and system availability constraints placed on hardware performing critical tasks. Providing increased redundancy at the root nodes allows for system degradation without loss of primary communication paths between the root and functioning leaf nodes of the system.

3.0 PRISM OPERATING SYSTEM

The general functional requirements of the PRISM operating system are designed to support decentralized embedded systems. In particular, integration of computing systems in an aircraft is possible through adoption of the PRISM system as a standard architecture. For this reason, system capabilities are driven by management of communication and faults for a large set of embedded systems.

PRISM is designed as a proof of concept system to support fault tolerance in distributed embedded systems. This places many general purpose computing needs beyond the scope of PRISM. For example, the PRISM operating system has no need to support large (hundreds of gigabytes) data storage, and tasks are never created after system elaboration.

Another virtue of avionics embedded systems is that prior to download of a test application, the distribution of Ada packages onto processors is efficiently determined. In addition, alternate processing sites for task migration upon failure can be specified at *system build* time. System build of a distributed Ada program includes compilation of source code, assignment of tasks to processors and linking together of required packages for each processor.

Maintaining adequate processing power in the presence of faults may require run-time reconfiguration of task assignments and require changes in communication routing. At system build time, initial and alternative task assignments are made. This is appropriate for embedded systems which often run tasks requiring specific resources such as sensors and actuators. Also, limiting the sites available for task processing simplifies the decentralized scheduling problem.

The following is a list of high level PRISM operating system requirements.

- Task scheduling, synchronization and other embedded system support is provided at each individual processing site.
- A central system interface is provided to download application tasks to any processor, begin their execution and monitor their status.
- A standard high level interface is supported for communication between tasks within a cluster as well as between tasks in separate clusters. The same underlying interface is used for both operating system and application communication.
- Communication routing paths are reconfigurable both statically at system build time and dynamically due to run-time reconfiguration.
- Fault management through voting on global data communication provides fault detection, isolation and recovery from both hardware and software faults.
- Global data and I/O access is supported throughout the system.
- The local operating system and network operating system combine to support programming embedded distributed embedded applications using Ada tasking calls.

The required capabilities of PRISM are provided through three decentralized sub-systems: the distributed Ada tasking executive (referred to as the *distributed executive*), network operating system and local operating system. Copies of each subsystem are provided in read only memory (ROM) at appropriate processing sites. The distributed executive and local operating system exist at each processor. The network operating system executes at each cluster controller.

The operating system is independent of PRISM architecture variation. Different numbers of processors in a cluster, different cluster inter-connections and different levels of cluster redundancy are all supported by the same versions of distributed executive, local operating system and network operating system. Any processor with a distributed executive and local operating system in ROM may be inserted into any cluster. The same is true of a cluster controller with network operating system in ROM.

Provided that an embedded application (referred to as an application) has the necessary local cluster resources, applications may be downloaded into any processor in any cluster. If the network operating system routing tables contain entries for all tasks accessed globally by an application, any processor in that cluster can complete the same inter-cluster, distributed tasking calls.

Ada run-time support is provided at each processor and cluster controller by a copy of the distributed executive. This distributed executive is derived from a validated single processor Ada Tasking Executive as described in Section 3.1. Modifications of the Ada tasking executive to become a distributed executive are not extensive.

Decentralized system control software allows system configuration and task assignment. Task down-loading to processors and routing table updates are control duties which require only data messages from a central control to network operating system subsystems. Start-up diagnostics, run-time status collection, error reporting, and post-fault re-configuration of task assignments all require decentralized control functions executing at each node. For this reason, in PRISM a single system cluster controller is chosen as the central control interface to an operator console. Other nodes contain system controllers which execute the distributed functions requested by central control. Central control may be migrated from one node to another for fault recovery.

Status and error reporting through system control forms a basis for distributed debugging tools. For a distributed embedded system to be implemented and tested, a decentralized tool for monitoring execution is needed. This tool interfaces the application engineer to the internal execution state of the system. Monitoring a single processor is a complex assignment. In order to minimize network transfers of state information, a decentralized system monitor should reduce the total number of system state changes reported.

In PRISM, a two stage debugging process is used. First, a decentralized debugging tool narrows the number of possible nodes in error. Next, an unobtrusive monitor can be attached to the processors within a node to record detailed state change information. This two step process provides for analysis of global state information followed by a narrowed observation of specific processing sites.

The network operating system provides a mailbox interface to both the local operating system and all sub-systems of the network operating system. This homogeneous interface provides operating system communication between network operating system and local operating system as well as supporting distributed tasking. System error messages and status are reported through this same mechanism. Priorities are assigned to each message packet and multiple out-of-order packets are managed. The local and global routing table schemes provide dynamic reconfiguration both within a cluster and between clusters (See Section 3.3).

Both redundant and non-redundant messages are supported through the global communication. This unified communication interface provides for fault tolerant system task redundancy as well as redundant application tasks.

Fault management is provided through hardware based detection and software based isolation, masking and re-configuration. The raw comparison of redundant communication data is provided by each redundancy unit. The management of redundancy units, sets of packet copies, detection of transient errors and maintenance of the configuration fault database is provided in network operating system software.

Management of Global data is under development. Data object managers which exist on the global network in replicated and non-replicated form are being considered. This would allow application and system tasks to share data through a message based transaction mechanism.

One goal of PRISM has been to support distributed Ada tasking as a high level language interface for application program communication. The distributed executive, local operating system and network operating system combine to allow an application task to rendezvous with any application task in any cluster. A more detailed discussion of this mechanism is provided in Section 3.4.

The initial implementation of the network operating system and local operating system uses the Ada tasking model. The use of Ada tasking has allowed rapid prototyping of the operating systems, although the operational speed is slower than desired.

3.1 Distributed Ada tasking executive

The primary purpose of an Ada tasking executive is to provide run-time support for one or more tasks, inter-task synchronization, communication between tasks, and external communication with other devices. This support is provided on the same processor that is executing the Ada program. To extend an Ada tasking executive to a distributed executive, support for synchronization and communication between tasks with physically separate Ada tasking executives is required. This occurs when an Ada program is executing on two or more processors each with its own Ada tasking executive.

The PRISM distributed executive is based on the Rockwell DDC Ada/CAPS compiler system [Ada 87] Ada tasking executive. This is a validated Ada compiler providing all facilities in the Ada Language Reference Manual [LRM 83]. This cross compiler is targeted for the Advanced Architecture Microprocessor. The interface between Ada program modules and the Ada tasking executive is based on a software trap mechanism allowing Ada tasking executive service routines to be executed in executive mode. The distributed executive provides extended facilities through expanding the trap service routines.

The primary difference between the distributed executive and the standard Ada tasking executive is the addition of service routines for creating remote tasks (both caller and callee), use of the local operating system to request a remote entry call, interface to the local operating system for receipt of remote tasking messages, and management of tasks which are awaiting completion of a remote entry call.

3.2 Local operating system

Each processor executes an independent set of application tasks supported by a local distributed executive. Each distributed executive is interfaced to the network operating system for remote task communication. This application task–to–network–operating–system interface supports other functions such as answering processor status queries, starting and reporting results of processor diagnostics, sending contents of processor memory and downloading application tasks to the processor. These interface duties are incorporated into the local operating system.

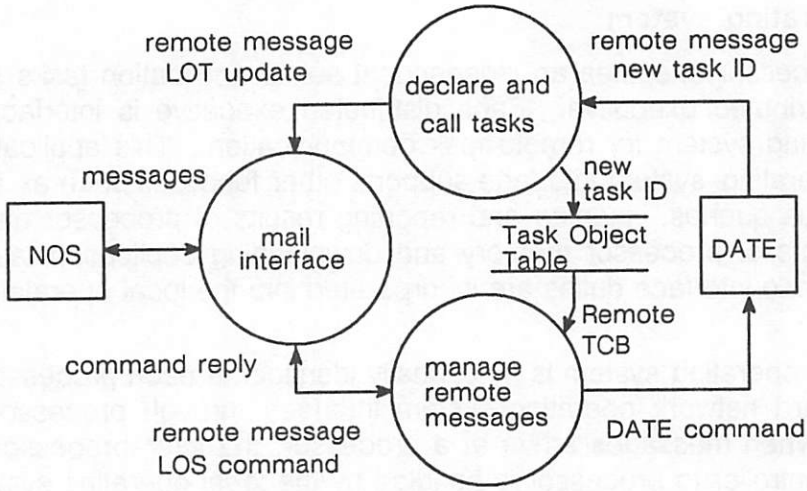
The local operating system is functionally identical at each processor. It provides a standard network operating system interface through processor mailbox procedures. When messages arrive at a processor, the inter–processor interrupt from cluster controller to processor is handled by the local operating system. The local operating system initiates the interrupt from processor to cluster controller that causes messages to be sent. Packets are buffered upon input and output by mailbox routines. When a message arrives, the packet is examined and classified into command categories.

The dataflow diagram of the local operating system is shown in Figure 3. When the incoming message is a distributed executive command, such as a rendezvous, rendezvous complete or rendezvous failure, a software trap transfers control to the distributed executive. The distributed executive then performs the corresponding executive service. If the incoming packet is a local operating system command, such as *binary download* or *get binary* from memory, the local operating system performs the appropriate action. The local operating system can use mail interface procedures to send results of local operating system commands or acknowledge receipt of messages.

When a rendezvous request is received by a local operating system, a trap is issued to the distributed executive. The task control block of the task to be called is passed as a parameter. This task control block is obtained from a table which is searched with the task destination object identifier as a key. This object identifier–to–task–control–block relationship is stored in the task object table at the time the called task is created by the distributed executive.

A task which performs a remote entry call must be created as a remote task even when it is not called from another processor. As illustrated in Figure 3, the task control block of such a task must be stored in the task object table so that the local operating system can give that task control block to the distributed executive when a rendezvous complete or rendezvous failure message is returned from the called task. A task control block is entered in the task object table when a remote task is created.

The task object table is used to obtain the task control block of the calling task when the rendezvous complete message is received by the local operating system. At that time a different trap is issued to the distributed executive. Between the sending of the rendezvous request by a distributed executive and the receipt of a rendezvous completed message, the calling task is treated like any task waiting on an entry call. The calling task is placed on a queue of tasks waiting for remote entry call completions.



command reply:	data in response to LOS command
DATE command:	remote rendezvous commands and replies
LOS command:	includes load Ada unit, get binary from memory
LOT update:	new task object ID entry into local object table
messages:	operating system and tasking packets
new task ID:	remote task id from DATE creation
remote message:	remote entry call or reply
remote TCB:	locally created remote task control blockpointer

Figure 3. Local Operating System Dataflow

3.3 Network operating system

The data flow between the two primary network operating system subsystems, regional message manager (referred to as the *regional manager*) and global message manager (referred to as the *global manager*), is shown in Figure 4. The regional manager and global manager constitute the network operating system communication subnetwork. This subnetwork provides inter-node and intra-node communication. The terms *node* and *cluster* are used interchangeably.

A message passing scheme with mailboxes stored in an regional memory is used to communicate between processors within the same node. Within a node, all network operating system subsystems and the processors communicate through the common mailbox interface.

When sending messages between nodes, an object oriented addressing scheme is used. Initially, messages are transferred to the nodes global manager (via the regional manager) which selects the output bus interface and sends the message. At each node in the route from source to destination, a new output bus is chosen.

Since multiple tasks can execute on a single processor site, more than one communication session may be open between any two processors. The network operating system transmits messages as packets which may follow the most convenient route from source to destination.

To aid in the movement of data and tasks between processors, an object oriented addressing scheme has been implemented. Local object tables, global object tables and path tables are maintained in each node.

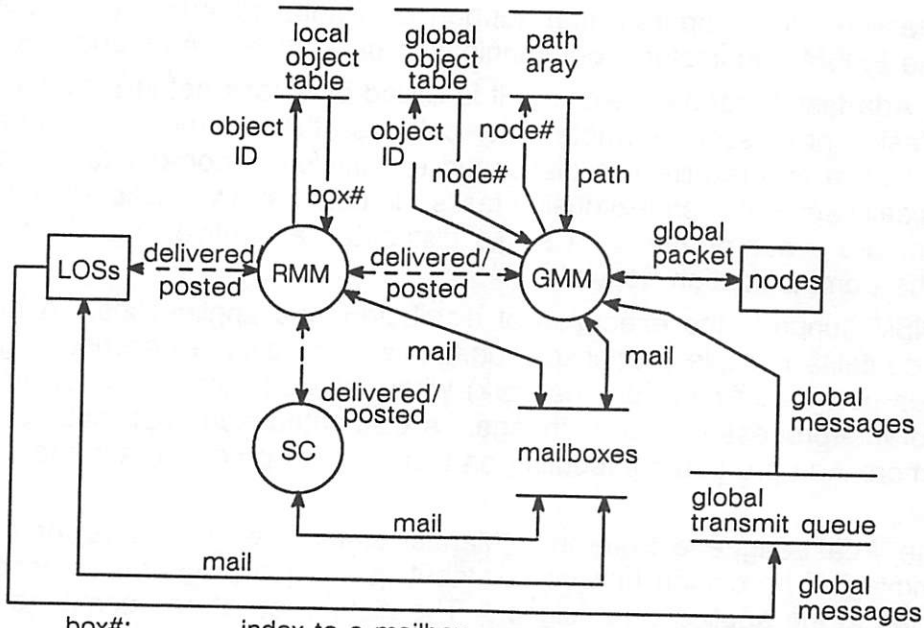


Figure 4. Network operating system dataflow

Local object tables contain the mapping between object identifiers (assigned at system build time) and local node mailboxes. At each node, this table guides the transfer of messages within a node.

When a message is transferred between nodes, the output bus from each node is calculated by table look-up. These routing tables are called the global object table and the path table. Although any two tasks may have only one active message exchanged at any one time, multiple tasks can communicate concurrently between two (or more) processors.

Global object tables are used to determine the destination node address of messages leaving the source node. The destination node address is used to access the path table which contains alternative output buses used to reach the destination. Alternative paths may be cycled through for bus fault avoidance and communication load balancing.

When an object is moved within a node, only the local object table of that node must be updated. When an object is moved between nodes, updates are performed on the global object tables of all nodes containing tasks accessing the moved object. This allows low overhead routing decisions and low cost migrations of objects.

3.4 Distributed Ada tasking

Support of Ada task communication for a distributed embedded systems requires special implementation of a decentralized operating system, distributed executive, and custom tools that use the decentralized system for configuration, start-

up and testing. It is apparent that support of distributed Ada programming influences the PRISM architecture, communication network and Ada language tools.

An Ada task is termed remote if it is called by a task not in residence on the remote task's processor. Remote entry calls use the communication network for synchronization and require two distributed executives to coordinate scheduling of the physically separate rendezvousing tasks. In contrast, two tasks within the same processor are synchronized by a single distributed executive, and do not require use of the communication network.

PRISM supports the execution of distributed Ada applications. A distributed system contains multiple processors communicating over a network (e.g. a bus, local area network or long haul network) where shared memory is not the primary means of interprocessor data exchange. A distributed Ada application consists of one or more Ada programs executing on two or more processors in the distributed system.

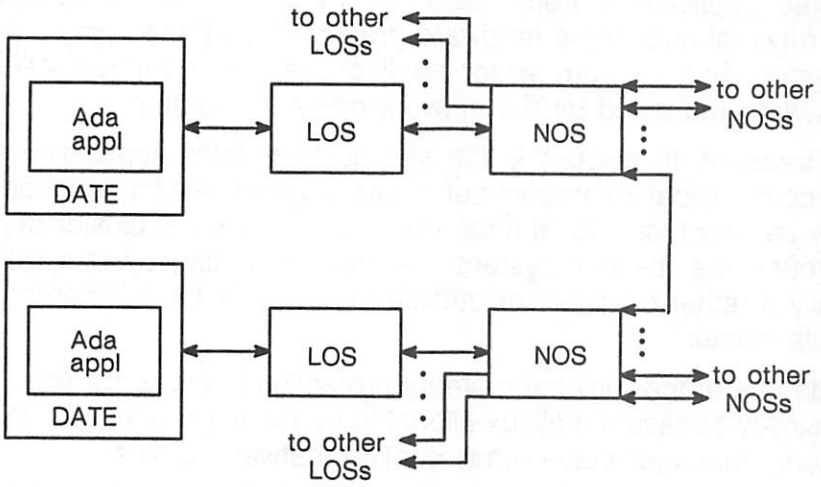
The Ada Language Reference Manual states few guidelines for distributed programming. The choice between distributing multiple programs or arbitrary Ada units is left to the application designer. The distribution of one complete Ada program allows the compiler to complete syntax and semantic checks across distributed Ada units. Restricting the application to using full Ada for tightly coupled processors, and allowing task entry calls to be the only network communication, removes difficult system support problems [Mudge 87, Tedd 84].

Tightly coupled processors (i.e. processors sharing memory) can execute tasks using data from the same library package. However, maintaining distributed programmer transparency to processor task assignment puts an extra burden on the operating system. This is because consistency of the shared data must be maintained no matter where that data resides. Data consistency can be accomplished through the implementation of a decentralized global data manager. Besides enforcing concurrent access controls, the data manager manages the mapping of structures to physical storage and interprets a global data addressing scheme. By limiting the interaction of distributed Ada units, the need for such a data manager is removed.

The network operating system and local operating system support only task entry calls for communication between distributed Ada units. It is not our goal to maintain transparency of program distribution where the system chooses the processing sites for each Ada unit. It is our goal to supply a standardized distributed programming interface that requires little effort to move a prototype system from a non-distributed development environment onto PRISM. This is most easily done through restricting tasks that share global data to execution on processors within the same cluster. The shared data is then conveniently stored on that cluster's regional memory.

As diagrammed in Figure 5, interaction between the distributed executive, local operating system and network operating system provide transparent remote task entry calls to the application. With proper automation of linking and compiler pragmas, the remote entry call looks identical to a local entry call. Rather than using language pragmas, a notation for distributed Ada programs such as APPL [Jha 89] would be useful. Such a notation centralizes the code *fragment* to *station* mapping into one specification and allows redundant mappings to be specified.

Tools for such pre-partitioning of Ada code with redundant entry calls have been developed [Hutcheon 88].



Ada appl: remote tasking application packages
DATE: distributed Ada tasking executive
LOS: local operating system
NOS: network operating system

Figure 5. PRISM operating system interaction

Of course, execution for a remote entry call takes longer than a local one. Code for the run-time entry call execution differs between remote and local, since remote calls require use of the network. An entry call between tasks on the same processor uses nothing but the distributed executive. An intra-node entry call uses the local operating system to local operating system message passing supplied by a network operating system regional message manager. For an inter-node entry call, the rendezvous request must pass through a local operating system, regional manager and network operating system global message manager at both the source and destination nodes. In addition the inter-node entry call may forward a rendezvous request through intermediate node global managers.

The distributed executive, local operating system and network operating system design structure parallels the design of ADME [DiGrazia 87]. The ADME *System Executive* is functionally similar to the system controller. The same is true for the ADME *Distributed Executive* and the local operating system. One major difference is the fault management support of PRISM. ADME relies on fault detection through a polling between processors while PRISM uses hardware voting for fault detection and masking.

4.0 SYSTEM IMPLEMENTATION TRADE-OFFS

Two alternative operating system designs were implemented. The first design proved that the communication subsystem functioned properly but required optimization. To increase communication throughput, operating system functions were distributed from the network operating system to the local operating systems, tasks were eliminated from the network operating system, concurrency control for shared objects was provided by locks, and interrupt connected procedures substituted for interrupt entry calls. The effect of these optimizations is quantified in Section 4.2.

4.1 Operating System Optimizations

Originally, the design of the network operating system and local operating system protected applications from interfering with each others communication. Access to the regional memory is hardware protected, thus enforcing a partitioning of regional memory between processor applications. All communication queues and mailboxes were managed by the network operating system.

The first trade-off to discuss is the sacrificing of inter-application protection for increased communication throughput. The original design required all messages, whether destined for a local node object or an external cluster object, to be transmitted through the mailbox system. A local operating system mailed messages identically whether destined for outside the node or for another local operating system in its cluster.

This made use of the regional memory protection in the mailboxes. No processor can physically access a mailbox allocated to another processor. When delivering, the regional manager copies mail packets between boxes.

One major optimization has been the decentralizing of the global message formatting and enqueuing. Routines for the formatting and enqueuing of global messages are included in every local operating system. Instead of the global manager performing this message management for all local operating systems in the cluster, each local operating system on a separate processor performs its own. This more equally distributes the message management processing load over the cluster.

The only signal between processor and cluster controller is a single interrupt line. After distributing global message preparation to local operating systems, this interrupt signals both posting of mail and enqueuing of global messages. When a processor-to-cluster-controller interrupt is serviced by the regional manager, a check of the global transmit queue is done (See Figure 4). If it is not empty, the task that controls output of messages to bus interfaces is awakened.

With the above optimizations, protection from applications interfering with each other is diminished. The transmit queue is placed in a multiple processor shared regional memory area. No hardware support is provided to keep an application task from circumventing the queue management or spoiling the queue memory. This constitutes a trade of protection for increases in communication throughput.

A related optimization was the elimination of as many Ada tasks from the operating system as possible. This included tasks which were providing nothing but concurrency control to shared data objects. The concurrency control tasks were used to control access to an object such as the global transmit queue. For example, an entry for *Enqueue*, *Dequeue* and *Wait_On_Empty_Queue* were provided in a *Transmit_Queue* task. This made a clean queue implementation but added milliseconds to each message transmission.

Through Wait_On_Empty_Queue synchronization calls to every shared queue object, only operating system tasks with data to process were on the distributed executive run queue. Consumer tasks with no data to process were waiting on the appropriate producer task's entry queue. This reduced the distributed executive

scheduling overhead incurred if tasks were periodically awakened, placed on the run queue, but had no work to do.

As an optimization these concurrency control tasks where replaced with packages containing subprograms for enqueue and dequeue. Concurrency control was implemented through *spin locks* based on small machine code *test and set* routines.

This removal of tasks had a major drawback. A task which consumed data from a queue was, in the previous implementation, blocked on an entry call when no data was in the queue. With the new method, such a consumer task had to either delay itself or find another means of blocking itself. An Ada delay can be an expensive waste of processing power. However, no other appropriate method of re-scheduling a task is available in Ada. The chosen solution was to create a small signal task for the consumer to call. This signal task was also called by a producer, such as the regional manager shown in Figure 4, enqueuing global messages. A dozen of these signal tasks exist in the network operating system. They are small, high priority tasks with only entries for synchronization rendezvous. They cause more overhead than a simple embedded executive supported semaphore construct.

The primary difference between this Ada implementation and a typical embedded embedded system is that, in the latter systems, simple *signal* and *wait* primitives would have been used for task synchronization. Although harder to debug, the signal and wait primitives require significantly less scheduling overhead than the tasking entry calls.

The last optimization to be discussed is that of interrupt service. Our original implementation made use of interrupt entry calls to service tasks. These tasks provided the hardware service, interrupt acknowledge and data transfer when an interrupt occurred. For example, the interrupt from a DMA-complete operation was handled by a procedure connected directly to that interrupt.

One problem with Ada is the loosely defined interrupt service. A compiler may define interrupt service connection by varying methods. For fast interrupt service of time critical devices, avoiding the Ada task scheduling mechanism is mandatory.

As suggested by the Ada Language Reference Manual, the distributed executive uses *interrupt entry calls* for handling interrupts. Interrupt entry calls require the handler to proceed through the normal scheduling mechanism. For many types of interrupt handlers, this type of scheduling is too slow. If many interrupt handler tasks are active at one point in time, they all have the highest system priority and compete for processing time. With these problems, interrupt entry calls are functionally adequate for handling bus interface, DMA and processor to cluster controller interrupts, although the additional scheduling overhead is undesirable.

Time critical RS232 interrupts for interface of the system console to the central controller require direct connection of interrupts to a handler procedure to avoid buffer overflow. Such a direct connection circumvents the normal scheduling mechanism and calls the connected procedure directly upon interrupt.

Another example is bus interface service. When a bus interface signals a message received, quick service of that bus interface entails reading the status, checking for errors and enqueuing a descriptor of the arrived message. A con-

sumer task of this message queue may be blocked waiting for data. Some method of synchronization between the bus interface interrupt service routine and the consumer task is necessary. Therefore, an entry call from an interrupt service procedure is mandatory.

In our current distributed executive, synchronization rendezvous from interrupt service procedures are not possible. The interrupt entry call uses the Ada scheduling mechanism and adds a millisecond to interrupt service. Thus, further work must be done to optimize the distributed executive support for network operating system interrupt service.

4.2 Empirical Comparison

A series of experiments were performed to provide insight into the relative increase in system throughput due to the above operating system optimizations. Average timings between synchronization rendezvous within a processor, between processors within the same cluster, and between processors on neighboring clusters were calculated.

For each set of experiments, from 1 to 12 tasks were activated by each distributed executive. A caller task and a corresponding callee task were activated to repeatedly rendezvous. For the single processor experiment, both caller and callee were created as normal tasks. For the intra-cluster and inter-cluster experiments, caller and callee were created as remote tasks by separate distributed executives. Timing results from these experiments were collected through the Rockwell Computer Development Station which implements an unobtrusive monitoring device.

The removal of tasks from the network operating system and distribution of global message functions to the local operating systems cause an increase in throughput for distributed rendezvous. Figure 6 shows the base rendezvous time for one processor running from 2 to 24 tasks. Figures 7 and 8 give the timing data for rendezvous between tasks on separate processors. The data in Figure 7 is from processors within the same cluster. The data in Figure 8 is from processors in neighboring clusters. Optimized timing plots represent the network operating system after the optimizations described in Section 4.1 were performed.

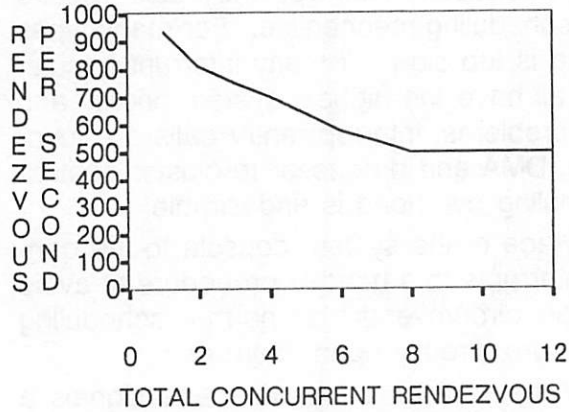


Figure 6. Single processor entry calls

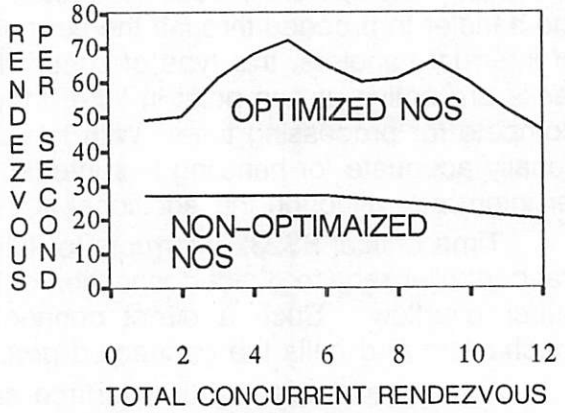


Figure 7. Intra-Cluster entry calls

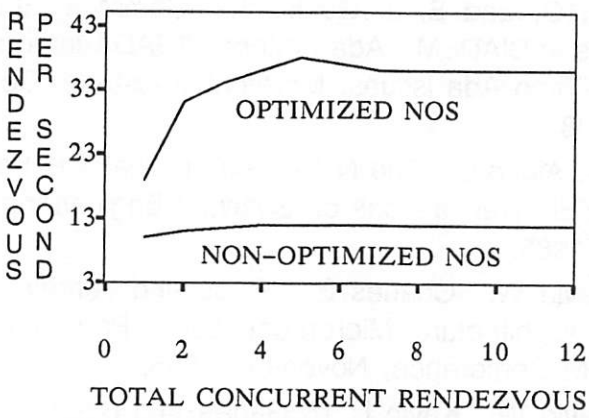


Figure 8. Inter-Cluster task entry calls

5.0 CONCLUSIONS

The PRISM system has provided a facility for testing the design and implementation of distributed processing support for embedded embedded systems. It is designed for fault tolerant multiprocessing of distributed Ada programs. The hardware and basic decentralized operating system successfully support remote Ada tasking. Ada tasking was used to prototype the operating system.

Optimizations to the Ada tasking executive used for the network operating system can reduce the remote entry call execution time dramatically. Optimizations to the network operating system have included distributing regional manager functions to each local operating system, distributing network packet formatting to each local operating system, and replacing entry calls with the use of low overhead spin locks for network operating system queue and table concurrent access control. With these optimizations, average distributed entry call rates were improved by 100% over earlier pure Ada tasking versions of the network operating system.

The use of full Ada for prototyping PRISM operating systems has allowed a fast, well documented implementation of the network operating system and local operating system. Our current Ada tasking overhead must be reduced to allow this type of programming to be used in production systems. Since the implementation of distributed Ada provides an intuitive interface for programming distributed embedded systems, this is a desirable goal.

In the future, the PRISM prototype system will be used as a testbed for the development of distributed embedded executives, redundancy management techniques, decentralized debugging tools, and decentralized scheduling mechanisms.

6.0 REFERENCES

[Ada 87] Ada Compiler Validation Summary Report: Rockwell International DDC-Based Ada/CAPS Compiler 1.0, VAX 11/118650 host and CAPS/AAMP target. Ada Joint Program Office, United States Department of Defense, Washington, D.C. AVF Control Number: AVF-VSR-112. 0288, June 1987.

[Atkinson 88] Atkinson, C. and S. J. Goldsack. Communication Between Ada Programs in DIADEM. Ada Letters, SIGADA International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, U.K, VIII(7), June 1988.

[Avizienis 85] Avizienis, Algirdas. The N-Version approach to Fault Tolerant Software. IEEE Transactions on Software Engineering, SE-11(12), December 1985.

[Best 85] Best, David W. Charles E. Kress and Jeffrey D. Russell. Advanced Architecture Microprocessor. Proceedings of the IEEE WESCOM Conference, November 1985.

[Best 88] Best, David W. Kevin L. McGahee and Roger K. Shultz. A Fault Tolerant Avionics Multiprocessing System Architecture Supporting Concurrent Execution of Ada Tasks. 8th Digital Avionics Systems Conference, San Jose, CA, October 17-20, 1988.

[DiGrazia 87] Joseph C. DiGrazia, Jehuda Ziegler and Richard Mueller. An Ada Distributed Multiprocessor Executive: From Conceptualization to Implementation. Ada Letters, ACM SIGADA International Conference, Boston, MA, December 1987.

[Hutcheon 88] A.D.Hutcheon and A. J. Wellings. Supporting Ada in a Distributed Environment. Ada Letters, SIGADA International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, U.K, VIII(7), June 1988.

[Jha 89] Rakesh Jha, J. Michael Kamrad II, and Dennis T. Cornhill. Ada Program Partitioning Language: A Notation for Ditributing Ada Programs. IEEE Transactions on Software Engineering, 15(3), March 1989.

[Knight 87] Knight, John C. and John I. A. Urquhart. On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems. IEEE Transactions on Software Engineering, SE-13(5), May 1987.

[LRM 83] Reference Manual for the Ada Programming Langurage. Washington, D. C. U.S. Dept. of Defense, 1985.

[Lyttle 85] Lyttle, D. IBM-PC Based Ada Symbolic Debugger, Proceedings of the Annual National Conference on Ada Technology, Washington, D.C. October 1985.

[McGahee 88] McGahee, Kevin L. PRISM: Fault Tolerant Data Communications for Multiprocessing Applications. Masters Thesis, Department of Computer Engineering, Iowa State University, Ames, IA, Spring 1988.

[Mudge 87] Mudge, Tevor. Units of Distribution for Distributed Ada. Ada Letters, ACM SIGADA International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, May 1987.

[Tedd 84] Tedd, Mike, Stefano Crespi-Reghizzi and Antonio Natali. Ada for multi-microprocessors. Cambridge University Press, London, 1984.

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Telephone: 415 528-8649
Email: office@usenix.org